



Stefano Berretti
Laura Carnevali
Enrico Vicario

Fondamenti di Programmazione

linguaggio c, strutture dati e algoritmi elementari, c++



Indice

Introduzione	1
1 Rappresentazione	3
1.1 Un frammento del linguaggio c	4
1.1.1 Tipi, variabili e costanti	4
1.1.2 Operatori ed espressioni	5
1.1.3 Istruzioni	5
1.2 Rappresentazione dei dati	14
1.2.1 Numeri	14
1.2.2 Interi senza segno	19
1.2.3 Caratteri	19
1.2.4 Interi con segno	20
1.2.5 Valori in virgola mobile	21
1.3 Rappresentazione delle istruzioni	27
1.3.1 Assembler	27
1.3.2 Linguaggio macchina	32
1.3.3 Esecuzione su un processore	36
1.3.4 Compilazione, assemblaggio e collegamento	44
1.4 Definizione di un linguaggio	47
1.4.1 Sintassi di un Linguaggio	47
1.4.2 Grammatica	47
1.4.3 Albero sintattico	50
1.4.4 Il metalinguaggio BNF	52
1.4.5 Verifica lessicale, sintattica e contestuale	54
1.4.6 Semantica e sintassi	55
1.5 Il linguaggio c	58
1.5.1 Tipi, variabili e costanti	58
1.5.2 Operatori ed espressioni	62
1.5.3 Puntatori	68
1.5.4 Array	76

1.5.5	Istruzioni	89
1.5.6	Funzioni	98
1.5.7	Dati strutturati	110
1.6	Esercizi	114
2	Strutture dati e algoritmi elementari	127
2.1	Liste	128
2.1.1	Rappresentazione in forma sequenziale	129
2.1.2	Rappresentazione collegata con arrays e indici	132
2.1.3	Rappresentazione collegata con puntatori	138
2.1.4	Iterazione e ricorsione	144
2.2	Alberi	153
2.2.1	Alberi binari di ricerca	154
2.2.2	Forma collegata con puntatori	155
2.2.3	Forma sequenziale	163
2.2.4	Forma collegata con indici	167
2.3	Costo di esecuzione e complessità	170
2.3.1	Il modello di costo	170
2.3.2	La complessità di un algoritmo	172
2.4	Algoritmi di ricerca	174
2.4.1	Ricerca sequenziale	174
2.4.2	Ricerca binaria	175
2.4.3	Ricerca a salti	177
2.4.4	Esercizi	179
2.5	Algoritmi di ordinamento	181
2.5.1	Sequential-sort	181
2.5.2	Bubble-sort	183
2.5.3	Merge-sort	186
2.5.4	Quick-sort	192
2.5.5	Heap-sort	199
2.6	Complessità minima di un problema	205
2.6.1	La complessità del problema dell'ordinamento	205
2.7	Esercizi	208
3	Estensione dal c al c++	231
3.1	Legame per riferimento e riferimenti costanti	231
3.2	Classi e oggetti	232
3.2.1	Definizione	232
3.2.2	Visibilità public, protected, private	234
3.2.3	Overloading	235
3.2.4	Creazione	236

3.2.5	Costruttori e distuttore	237
3.2.6	Riferimento	240
3.2.7	Self-reference	240
3.3	Attributi e metodi statici	242
3.4	Ereditarietà	244
3.4.1	Ereditarietà di implementazione	244
3.4.2	Ereditarietà in modo public, protected e private	246
3.4.3	Ereditarietà dei costruttori	247
3.5	Polimorfismo e dynamic binding	247
3.5.1	Override e polimorfismo	247
3.5.2	Dynamic binding	249
3.5.3	Sostituibilità e metodi virtuali	251
3.5.4	Il problema della classe di base fragile	253
3.5.5	Classi astratte e interfacce	255
3.5.6	Ereditarietà multipla	257
3.6	Ulteriori letture	259
Testi di approfondimento		261

intesa come condizione necessaria per uno studio che dia spazio all'attenzione del lettore. Per questo è scritto senza intento di completezza, a tratti in modo anche rude, cercando di esporre i concetti in modo semplice ma senza rinunciare ad osservarne e discuterne la complessità.

Capitolo 1

Rappresentazione

Il problema della rappresentazione consiste nel dare ad un algoritmo una forma che possa essere eseguita su un elaboratore. In questo assume un ruolo centrale l'uso di un linguaggio di programmazione.

In questo testo affrontiamo il problema facendo riferimento al caso del linguaggio c. Questo ha un rilievo preminente in una ampia varietà di contesti applicativi, e costituisce comunque la base di c++ e java.

Dopo una prima breve descrizione di un frammento del c orientata a fornire un'intuizione circa le caratteristiche del linguaggio, la trattazione è ampliata in due direzioni opposte. Da un lato sono descritti i principi del processo che permette di tradurre un programma c in una sequenza numerica che possa essere caricata nella memoria di un elaboratore ed eseguita da un processore. Questo permette di intuire aspetti del processo di compilazione che determinano larga parte delle regole del linguaggio. Dall'altro sono introdotti alcuni elementi teorici che permettono di descrivere il linguaggio attraverso un nucleo di regole generali, capaci di cogliere in maniera compatta e univoca l'organizzazione sintattica e i contenuti semantici del linguaggio.

Facendo leva sui due diversi strumenti di concretezza e astrazione, viene finalmente affrontato in maniera dettagliata la descrizione di sintassi e semantica del linguaggio c.

1.1 Un frammento del linguaggio c

1.1.1 Tipi, variabili e costanti

Il c, come qualunque altro linguaggio, rappresenta dati secondo tipi. Un tipo è caratterizzato dai valori che può rappresentare e dalle operazioni che è possibile eseguire sui suoi valori. I tipi elementari del c sono `int`, `float`, e `char` che approssimano rispettivamente numeri interi, numeri razionali e caratteri. Non esistono valori Booleani, che vengono sostituiti dai numeri interi sotto la convenzione che 0 codifica un falso e qualunque valore diverso da 0 codifica un vero.

Una variabile è una locazione di memoria che contiene un valore di un tipo. Il valore può variare nel corso della computazione, mentre il tipo è invariante. In un linguaggio simbolico, come è il c, una variabile è associata anche a un nome che permette di riferirsi alla variabile senza doverne specificare l'indirizzo fisico in memoria. In c, il nome di una variabile è una qualunque sequenza di caratteri alfabetici, numerici, con possibile uso del segno di underscore '_'. Il primo carattere non può essere un numero. La sequenza ha una lunghezza arbitraria anche se il compilatore non distingue tra nomi che non differiscono sui primi 32 caratteri.

Ogni variabile deve essere dichiarata prima dell'uso, specificandone il nome e il tipo. Ad esempio, le dichiarazioni:

```
int count;  
float sum;
```

dichiarano due variabili di tipo intero e floating point a cui sono associati i nomi `count` e `sum`, rispettivamente.

Una variabile array è un insieme di variabili dello stesso tipo che possono essere referenziate attraverso un nome collettivo e un indice che le identifica. Da un punto di vista fisico è utile sapere che tali variabili sono memorizzate in locazioni contigue della memoria, anche se questo può restare trasparente al programmatore.

```
float A[100];
```

dichiara una variabile array di nome `A`, costituita di 100 variabili floating point. Le singole locazioni sono referenziate attraverso un indice intero entro parentesi quadre che denota l'offset della particolare locazione dall'inizio dell'array: `A[0]` denota la prima locazione, `A[4]` la quinta, `A[99]` l'ultima.

Una costante è un valore (di un qualche tipo) che non cambia nel corso della computazione.

1.1.2 Operatori ed espressioni

Un'espressione è una combinazione di costanti e variabili attraverso operatori.

Gli operatori sono usualmente classificati per tipo di operazione, distinguendo tra l'altro operatori aritmetici, relazionali, e logici. Gli operatori aritmetici sono la somma (+), la differenza (-), il prodotto (*), la divisione (/), il modulo (%). Gli operatori relazionali sono gli operatori usati per comparare valori: il minore (<), il minore o uguale (<=), l'uguaglianza (==), il maggiore o uguale (>=), il maggiore (>), il diverso (!=). Gli operatori logici sono la congiunzione (&&), la negazione (!), la disgiunzione (||). L'assegnamento (=) è anch'esso un operatore e l'assegnamento del risultato di una espressione a una variabile è a sua volta un'espressione.

Il calcolo di una espressione restituisce un valore e produce un effetto sui dati.

Il valore restituito da operatori aritmetici è quello che uno si aspetta: $a+5/b$ restituisce la somma di a con il risultato della divisione di b per 5 (a parte dettagli che per ora trascuriamo circa gli effetti di overflow e arrotondamento associati ai diversi tipi).

È notevole il caso dell'assegnamento che produce un effetto sui dati e restituiscue un valore: $a=b+125$ assegna ad a il valore di b aumentato di 125 e restituisce il valore che è assegnato ad a . $a=b+(c=125)$ assegna ad a il valore di b aumentato del valore restituito dall'espressione $c=125$; a sua volta tale espressione assegna il valore 125 a c e restituisce il valore 125.

Non è intuitivo il caso delle espressioni relazionali: $a>b$ restituisce 1 se l'espressione è vera e 0 altrimenti. Lo stesso vale per qualunque operatore relazionale e per gli operatori logici. Il fatto che gli operatori relazionali restituiscano valori interi per codificare il vero o il falso permette di scrivere anche espressioni mostruose, di limitato uso pratico: $a>=5$ restituisce 1 se a è maggiore o uguale di 5, e 0 altrimenti; dunque $a=(4<3)$ assegna 0 ad a , mentre $a=(3<4)$ gli assegna un valore diverso da 0, tipicamente 1.

L'effetto sui dati prodotto dal calcolo di una espressione è chiamato side effect (effetto collaterale). Mentre il termine suggerisce l'idea di una questione minore, fastidiosa ma non evitabile, i side-effects sono una componente strutturale dell'intero linguaggio. L'assegnamento, su cui in ultimo poggia qualunque elaborazione, è un particolare side-effect.

1.1.3 Istruzioni

Le istruzioni servono a dirigere il flusso dell'esecuzione: in c i dati sono elaborati attraverso i side effects prodotti dal calcolo delle espressioni, e il ruolo delle istruzioni è quello di determinare la sequenza con cui le espressioni sono eseguite.

Il fatto che il significato delle istruzioni risieda nel controllo del flusso fa sì che diventi conveniente descriverle usando i flow charts. Questi non hanno alcuna utilità pratica nella progettazione di un algoritmo, ma hanno il pregio di evidenziare in maniera intuitiva il modo con cui le diverse istruzioni sequenzializzano l'esecuzione delle espressioni.

Espressione, sequenza e compound

La più semplice istruzione è un'espressione seguita da un punto e virgola:

```
a=b+125;
```

L'esecuzione consiste nel calcolare l'espressione, producendone i side effects, e poi procedere alla istruzione successiva. Fig.1.1a illustra il concetto usando il formalismo dei flow-charts.

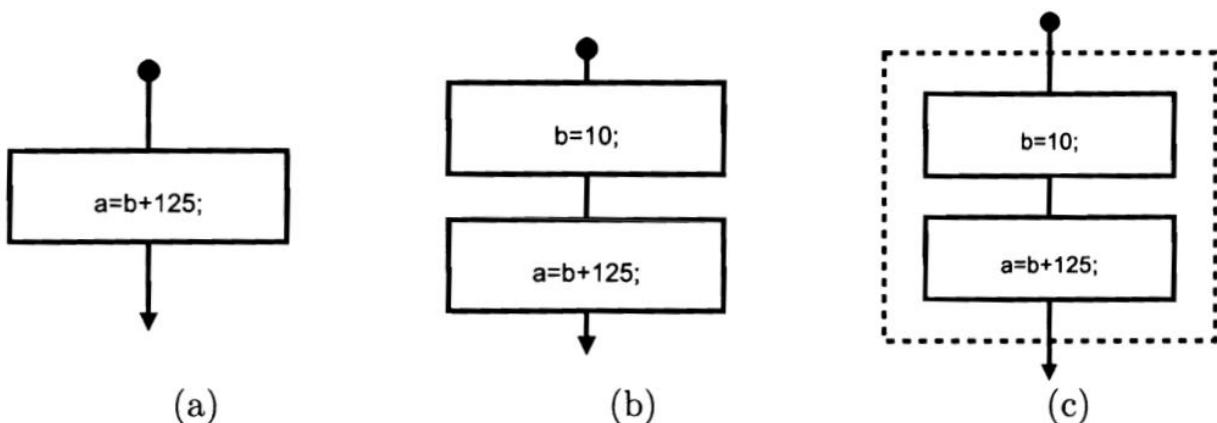


Figura 1.1: Rappresentazione in flow-chart di istruzioni elementari del c: **a)** istruzione semplice; **b)** sequenza di due istruzioni; **c)** sequenza di due istruzioni raccolte in un compound.

Due istruzioni in sequenza sono a loro volta un'istruzione. Ad esempio

```
b=10;  
a=b+125;
```

è una istruzione, la cui esecuzione è ottenuta eseguendo prima `b=10;` e poi `a=b+125;` (vedi Fig.1.1b). Si noti che la ripetizione della composizione in sequenza permette di avere una istruzione composta di un qualsiasi numero di istruzioni. A parte alcuni dettagli di rappresentazione legati all'uso di funzioni questo permette di riguardare un intero programma come un'istruzione.

Un'istruzione raccolta entro parentesi graffa è un'istruzione compound. Ad esempio:

```
{ b=10;  
  a=b+125;  
}
```

L'esecuzione del compound consiste nell'esecuzione dell'istruzione che sta entro le parentesi (vedi Fig.1.1c). L'utilità, non immediatamente apparente, è quella di una parentesi: raccogliere istruzioni in blocchi in modo che siano trattati in maniera unitaria. Questo in ultimo agisce sulla precedenza con cui sono associate le istruzioni successive e diventa rilevante rispetto alle istruzioni di controllo condizionale e di iterazione.

Condizione

Le istruzioni condizionali permettono di decidere direzioni diverse nel flusso di esecuzione in base al valore restituito da un'espressione.

L'istruzione condizionale **if** condiziona l'esecuzione di una istruzione, detta *corpo*, al risultato restituito da una espressione, detta *guardia*:

```
if(a>b)  
{  
  b=10;  
  a=b+125;  
}
```

L'esecuzione, è illustrata in Fig.1.2a: l'espressione di guardia $a > b$ viene valutata; se restituisce un valore diverso da 0 (un vero), allora viene eseguita l'istruzione $\{b=10; a=b+125;\}$; in ogni caso l'esecuzione prosegue all'istruzione successiva. L'esempio in Fig.1.2b illustra come le parentesi graffe del compound permettono di distinguere l'istruzione riportata sopra rispetto all'istruzione:

```
if(a>b)  
  b=10;  
  a=b+125;
```

L'istruzione condizionale **if** può essere estesa con una clausola **else** che permette di avere due corpi alternativi, eseguiti a seconda che la guardia restituisca un valore vero o falso (vedi Fig.1.2c):

```
if(a>b)  
  a=b+125;  
else  
{ b=10;  
  a=b+125;  
}
```

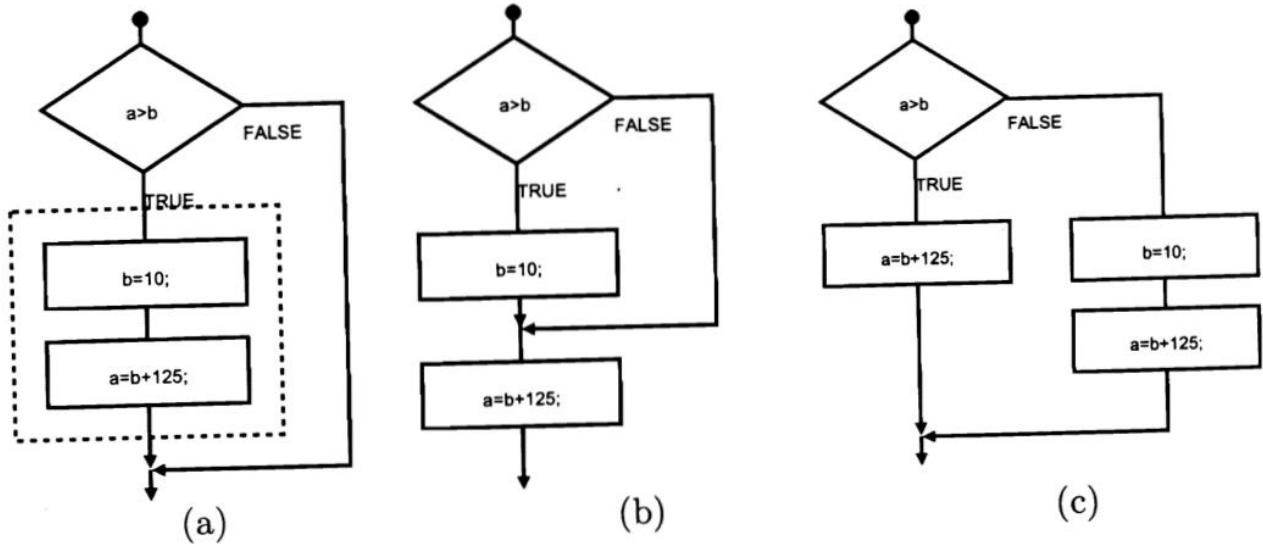


Figura 1.2: Rappresentazione in flow-chart delle istruzioni condizionali: **a)** if semplice; **b)** una variante del caso a) che illustra l'effetto del compound; **c)** if-else.

Iterazione

Le istruzioni di iterazione (loop) permettono di eseguire ripetitivamente un corpo di istruzioni fino al verificarsi di una qualche condizione sui valori delle variabili del programma. Esistono tre diverse istruzioni di iterazione: **for**, **while** e **do-while**, il cui diverso significato è illustrato in Fig.1.3.

Nell'istruzione **for**, un corpo di istruzioni è eseguito ripetitivamente sotto il controllo di una guardia costituita di tre espressioni di *inizializzazione*, *guardia* e *incremento*:

```
sum=0;
for(count=0;count<10;count=count+1)
    sum=sum+count;
```

L'esecuzione è illustrata in Fig.1.3a. L'espressione di inizializzazione è eseguita inizialmente, una sola volta. Segue una fase di iterazione nella quale, ad ogni iterazione: viene testata l'espressione di guardia; se la condizione è verificata viene eseguito il corpo, poi l'espressione di incremento, e infine si torna al punto di ingresso dell'iterazione; se invece la guardia restituisce falso, l'iterazione termina e il controllo passa all'istruzione successiva. Si osservi che è possibile il caso in cui l'espressione di condizione risulta falsa al momento della prima esecuzione e il controllo passa all'istruzione successiva senza avere mai eseguito il corpo.

Nell'istruzione **while**, un corpo di istruzioni è eseguito ripetitivamente fintanto che (mentre) una espressione di guardia restituisce un valore vero:

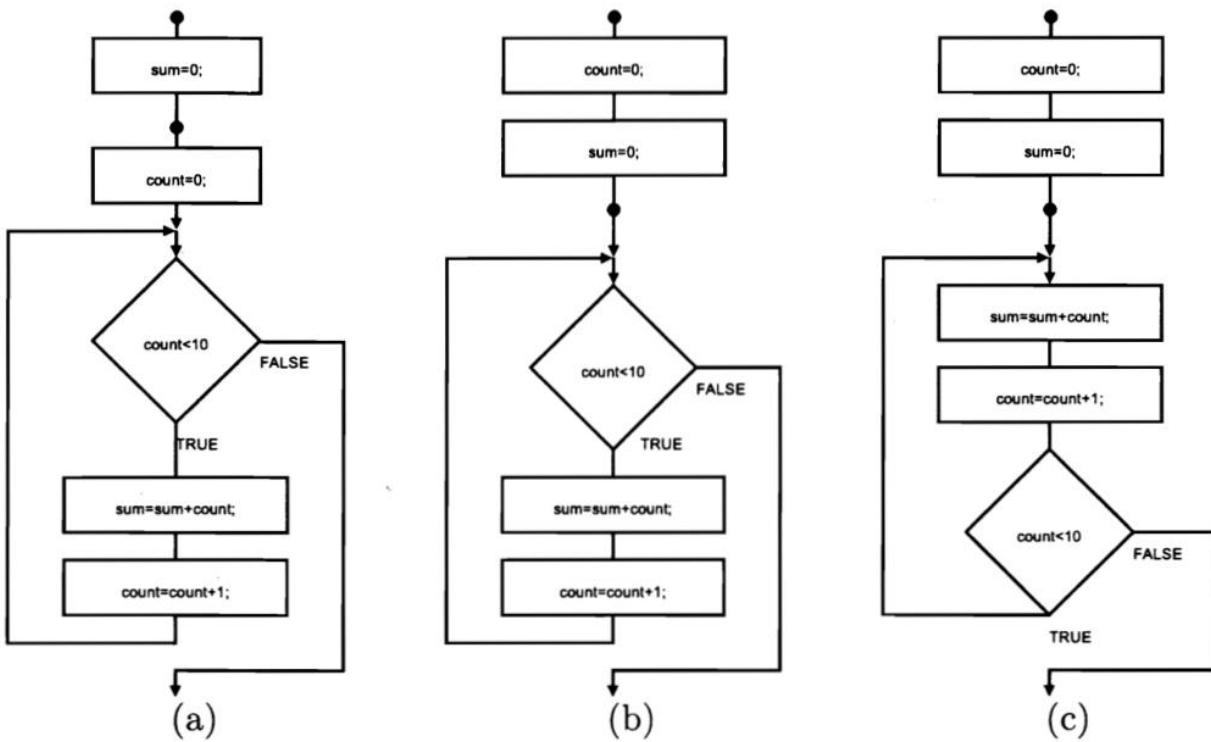


Figura 1.3: Rappresentazione in flow-chart delle istruzioni di iterazione del c:
a) for; **b)** while; **b)** do-while.

```

count=0;
sum=0;
while(count<10)
{
    sum=sum+count;
    count=count+1;
}

```

L'esecuzione è illustrata in Fig.1.1b: la guardia `count<10` viene eseguita; se il valore restituito è vero, viene eseguito il corpo `{sum=sum+count; count=count+1;}` e l'esecuzione torna all'ingresso della guardia. Quando la guardia restituisce falso il controllo passa all'istruzione successiva.

L'istruzione `do-while` opera in maniera analoga a `while` senonché in questo caso la condizione di guardia è posta in coda piuttosto che in testa al corpo:

```

count=0;
sum=0;
do
{
    sum=sum+count;
    count=count+1;
}while(count<10);

```

L'esecuzione è illustrata in Fig.1.1c: prima viene eseguito il corpo `{sum=sum+count; count=count+1;}`, e poi viene testata la guardia `count<10`; se la guardia restituisce un valore vero si torna a eseguire il corpo, altrimenti il controllo passa all'istruzione successiva.

Esempi

Gli elementi del C fino qui descritti, con l'aggiunta di qualche dettaglio, permettono di comporre alcuni programmi elementari.

Consideriamo il calcolo del fattoriale di un numero `n`. Una variabile moltiplicatore `fact` viene inizializzata a 1 e poi viene moltiplicata per tutti i numeri interi compresi tra 2 e `n`. Lo realiziamo usando un `for`:

```
main()
{ int n;           // dichiarazione di tre variabili intere
  int count;       // denominate n, count e fact
  int fact;

  n=10;            // 10 e' il valore di cui calcoliamo il fattoriale

  fact=1;
  for(count=2;count<=n;count=count+1)
    fact= fact*count;

  /* al termine dell'iterazione,
     fact contiene il fattoriale di n */
}
```

L'esempio introduce alcuni dettagli che sono per ora di minore rilievo, ma che risultano essenziali per il perfezionamento di un programma.

Il programma si apre con `main()` ed è interamente contenuto tra due parentesi graffe. Il significato di questo diventerà chiaro con la introduzione delle funzioni che permettono di scomporre il programma in sottoprogrammi. Per ora diciamo che `main` è il nome di una funzione e che questo è indicato dalle parentesi tonde che lo seguono. Le parentesi graffe delimitano il corpo della funzione, contenendo dichiarazioni di variabili e istruzioni.

I simboli `/* e */` sono delimitatori di commento: tutto quello che racchiudono è irrilevante ai fini dell'esecuzione del programma perché viene eliminato prima che il programma sia processato dal compilatore per produrre un codice eseguibile. In maniera alternativa, il commento può essere espresso a seguito del simbolo `//` che dichiara come commento tutto quello che appare fino alla fine della linea. In questo caso, un commento che si estende su più linee richiede che il simbolo `//` sia ripetuto in ciascuna di esse.

Spazi ripetuti, indentazioni, e linee vuote, sono essenziali per la leggibilità del programma, ma non sono rilevanti ai fini della logica del programma. In questo senso, il pezzo di codice riportato sopra è equivalente ad una unica stringa in cui sono soppressi i commenti, gli spazi ripetuti e le righe vuote.

Raffiniamo il programma per permettere il calcolo del fattoriale di un qualsiasi numero e per restituire il valore calcolato. Questo richiede operazioni di I/O (Input/Output) per l'acquisizione del valore di *n* e per la stampa di *fact* al termine del programma:

```
#include<stdio.h>
main()
{ int n;
  int count;
  int fact;

  printf("introdurre il valore di n"); //output a video
  scanf("%d",&n);                   //input da tastiera

  fact=1;
  for(count=2;count<n;count=count+1)
    fact= fact*count;

  printf("il fattoriale di %d e' %d",n,fact);
}
```

L'esempio include varie cose su cui per il momento è opportuno non applicare troppa attenzione: *printf* e *scanf* sono nomi di funzioni, il che è ancora indicato dal fatto che sono nomi seguiti da parentesi tonde. Il loro uso diventerà chiaro nella trattazione delle funzioni. Per il momento limitiamoci a osservare che le due funzioni sono definite da qualche parte, in maniera analoga al modo con cui sopra abbiamo definito la funzione *main*. Diciamo anche che *printf* stampa a video stringhe di testo e valore di variabili, mentre *scanf* assegna acquisisce da tastiera il valore di una o più variabili. In entrambe le funzioni, quello che sta tra le virgolette definisce il formato di quello che viene stampato a video o letto da tastiera; *%d* ha a che fare con il fatto che si stanno leggendo o scrivendo numeri interi; sarebbe un *%f* se si trattassero numeri floating point, o un *%c* se si trattassero caratteri.

L'uso delle funzioni *printf* e *scanf* richiede che in testa al programma sia aggiunta la direttiva di preprocessing *#include<stdio.h>* che fornisce la traccia per identificarne la definizione.

L'esempio include anche il simbolo *&*, prefisso a *n* nella chiamata alla funzione *scanf*, il cui significato diventerà chiaro con la descrizione dei puntatori e del loro uso nella comunicazione tra funzioni attraverso il legame dei parametri.

Come esempio ulteriore consideriamo il calcolo della somma dei valori memorizzati su un'array: in maniera duale al caso del fattoriale, un sommatore è inizializzato a 0 e poi incrementato con i valori di ciascun elemento dell'array. Il numero di elementi dell'array, e quindi il numero di iterazioni da eseguire, sono noti dalla prima iterazione. Anche in questo caso la struttura di controllo può essere creata convenientemente usando un **for**:

```
main()
{ float A[100]; // A e' un array di 100 numeri di tipo float
  int count;
  float sum;
  ...
  sum=0;
  for(count=0;count<100;count=count+1)
    sum=sum+A[count];
}
```

La stessa computazione può essere realizzata con l'istruzione **while**. In questo caso, la condizione di guardia `count<100` viene codificata nell'istruzione, mentre le espressioni di inizializzazione (`count=0`) e di incremento `count=count+1` sono codificate in due istruzioni aggiuntive:

```
main()
{ float A[100];
  int count;
  float sum;
  ...
  sum=0;
  count=0;
  while(count<100)
  { sum=sum+A[count];
    count=count+1;
  }
}
```

Infine, consideriamo la realizzazione con l'istruzione **do-while**:

```
main()
{ float A[100];
  int count;
  float sum;
  ...
  sum=0;
  count=0;
  do
  { sum=sum+A[count];
```

```
    count=count+1;  
}while(count<100);  
}
```

È utile osservare che il programma che abbiamo scritto è in ultimo un file di testo, che non è di per sé *eseguibile* più di quanto non lo sia una lettera in formato testo o un'immagine in formato jpg. Per farne un oggetto che possa essere caricato in memoria e eseguito su una CPU occorre che il file sia processato prima da un compilatore e poi da un assemblatore, in modo da ottenere una codifica numerica di dati e istruzioni.

La comprensione dei principi di questa traduzione permette di motivare le regole di un linguaggio, distinguendo elementi di disciplina intenzionalmente introdotti dai progettisti del linguaggio e limiti intrinseci del funzionamento di una macchina capace di eseguire un programma memorizzato.

1.2 Rappresentazione dei dati

Esistono due tipi di dati che occorre rappresentare nella codifica di un programma: i tipi che codificano testo e quelli che codificano valori numerici. Che sono in ultimo i due tipi di informazione che la nostra cultura ci insegnava a scrivere e riprodurre.

In C esistono tre diversi tipi dedicati a rappresentare valori numerici: `int`, `float` e `double`. I caratteri sono rappresentati con il tipo `char`, che poi è esso stesso codificato nella forma di un numero.

1.2.1 Numeri

Un numero è un ente dotato di un suo significato intrinseco, del quale è possibile dare rappresentazioni diverse. Il significato di un numero, almeno quello di un intero, può essere definito operativamente in riferimento alla relazione di equinumerosità tra insiemi: in questa prospettiva, il numero quattro trae il suo significato dalla numerosità di una particolare classe di insiemi equinumerosi, che siano quattro pere o quattro mele.

Una volta stabilito che quattro definisce un concetto con un suo proprio significato, di questo concetto è possibile dare rappresentazioni diverse: ci è usuale rappresentarlo con la cifra 4, ma diamo lo stesso significato alla rappresentazione `IV` del sistema di numerazione romana, o al 100 del sistema di numerazione binaria.

In generale, le diverse diverse rappresentazioni sono caratterizzate dalla combinazione di più convenzioni: la posizionalità della codifica, la base di numerazione, il numero di cifre, la codifica del segno, la rappresentazione di parti frazionarie e valori razionali.

Codifica posizionale

I numeri naturali sono codificati attraverso una base finita di cifre elementari. Nella esperienza quotidiana sono le cifre da 0 a 9 che formano la cosiddetta base decimale e codificano direttamente i primi 10 valori dell'insieme (dallo 0 al 9). I numeri successivi sono codificati componendo più cifre secondo una convenzione posizionale. Il che significa che il peso delle cifre dipende dalla posizione in cui appaiono: 125 e 215 codificano valori diversi.

In generale, se B è la base di numerazione di una codifica posizionale, la sequenza di cifre $a_N \dots a_1 a_0$, codifica il valore che può essere derivato attraverso il seguente *sviluppo polinomiale*:

$$[a_N \dots a_1 a_0]_B ::= \sum_{n=0}^N a_n B^n$$

a_N e a_0 sono rispettivamente la cifra più significativa (most significant digit) e la meno significativa (least significant digit)

Base di Numerazione

La base di numerazione B a cui siamo abituati è la base decimale. Il che è un fatto del tutto accidentale visto che l'evoluzione della specie su questo pianeta non è stata orientata a selezionare specie con capacità di calcolo ma piuttosto di manipolazione.

L'elaboratore usa la base 2. Anche questo potrebbe apparire come un accidente dell'evoluzione tecnologica, legato al fatto che in elettronica si realizzano bene dispositivi bi-stabili, in grado di rappresentare due valori diversi. Invece la base 2 ha una caratteristica che la rende unica e non accidentale: è la base minima che include cifre diverse. In sostanza è la base minima su cui è possibile contare in maniera posizionale. E tra l'altro è quella che richiede di studiare il minimo numero di tabelline: quella dello 0 e quella dell'1, entrambe banali. Il che semplifica un numero di algoritmi che devono essere realizzati nell'unità aritmetico-logica di un processore per eseguire operazioni quali la divisione, la moltiplicazione, la sottrazione.

Conversione della base di rappresentazione

Il fatto che noi esprimiamo valori usando la base decimale mentre l'elaboratore li rappresenta in forma binaria rende importante considerare il modo in cui la rappresentazione di un numero può essere convertita dalla base decimale a quella binaria e viceversa. Per lo più questa conversione rimane nascosta al programmatore. Ma capita che alcuni dati siano rappresentati in forma binaria. È spesso il caso di indirizzi di memoria, che possono apparire nell'uso di un debugger o nella configurazione di un dispositivo periferico.

Esistono due algoritmi di conversione diversi, basati sull'impiego dell'aritmetica della base di partenza o di quella di arrivo.

Per eseguire i conti nella *base di arrivo*, il numero iniziale viene sviluppato in forma polinomiale nella base di partenza, e i singoli coefficienti e potenze che ne risultano sono poi convertiti nella base di arrivo; la conversione è completata effettuando somme e prodotti nella base di arrivo:

$$[125]_{10} = [1 * 10 * 10 + 2 * 10 + 5]_{10} = [1 * 1010 * 1010 + 10 * 1010 + 0101]_2$$

Somme e prodotti tra numeri a più cifre sono calcolati come ci hanno insegnato a scuola. Con minore sforzo di memoria visto che qui basta conoscere la tabellina dello 0 e quella dell'1, entrambe banali. Ad esempio:

$$\begin{array}{r}
 1010 * 1010 \\
 \hline
 0000 \\
 1010 \\
 0000 \\
 1010 \\
 \hline
 1100100
 \end{array}$$

Sviluppando il conto, si ottiene dunque:

$$[125]_{10} = [1100100 + 10100 + 0101]_2 = [1111101]_2$$

Per eseguire il conto in base di arrivo occorre conoscere: come le cifre delle base di partenza sono rappresentate nella base di arrivo; come è rappresentata la base di partenza nella base di arrivo; come si fanno prodotti e somme nella base di arrivo. Si capisce allora che i conti conviene farli in base di arrivo quando la conversione trasforma la rappresentazione da base 2 a base 10. In questo caso la rappresentazione delle cifre binarie e della base 2 in base 10 è immediata, e le operazioni di somma e prodotto sono note dalla scuola elementare. Ad esempio:

$$[1100100]_2 = [1 * 2^6 + 1 * 2^5 + 1 * 2^2]_{10} = [64 + 32 + 4]_{10} = [100]_{10}$$

L'algoritmo dei resti successivi permette di effettuare la conversione operando nella *base di partenza*. Il che è pratico per conversioni da base 10 a base 2.

Assumiamo che A sia un numero e supponiamo di conoscerne la rappresentazione in base 10. La rappresentazione di A in base 2 è una sequenza di cifre binarie incognite $\dots, 0, 0, a_N, a_{N-1}, \dots, a_1, a_0]_2$ che è univocamente determinata dall'equazione

$$A = \sum_{k=0}^{\infty} a_k 2^k = a_0 + 2 \sum_{k=0}^{\infty} a_{k+1} 2^k$$

L'equazione implica che a_0 vale 1 se e solo se A è dispari, per cui possiamo determinare a_0 come il resto della divisione intera $A/2$. Una volta determinato a_0 , possiamo sostituire A con il quoziente della divisione $A/2$, che per definizione è uguale a

$$A/2 = \sum_{k=0}^{\infty} a_{k+1} 2^k$$

e procedere analogamente per determinare a_1 . Il procedimento termina quando il quoziente della divisione è 0: da lì in poi qualsiasi ulteriore divisione restituirebbe sempre quoziente 0 e resto 0. L'operazione è eseguita convenientemente organizzando i quozienti e i resti lungo una diagonale. Ad esempio, nella conversione del numero rappresentato in forma decimale dal 125, la sequenza dei quozienti è 125,62,31,15,7,3,1,0 e la sequenza dei resti è 1,0,1,1,1,1,1:

$$\begin{array}{r}
 125 \mid 2 \\
 124 \mid \cdots \\
 1 \mid 62 \mid 2 \\
 62 \mid \cdots \\
 0 \mid 31 \mid 2 \\
 30 \mid \cdots \\
 1 \mid 15 \mid 2 \\
 14 \mid \cdots \\
 1 \mid 7 \mid 2 \\
 6 \mid \cdots \\
 1 \mid 3 \mid 2 \\
 2 \mid \cdots \\
 1 \mid 1 \mid 2 \\
 0 \mid \cdots \\
 1 \mid 0
 \end{array}$$

La sequenza dei resti sono i coefficienti a_k in ordine inverso, dal Least Significant Bit a_0 al Most Significant Bit a_n . Nell'esempio: $[125]_{10} = [1111101]_{10}$

Base esadecimale

I numeri in base 2 sono lunghi ed è facile sbagliarsi nel leggerli e nel copiarli: memorizzare la rappresentazione $[125]_{10}$ è banale, ma $[1111101]_2$ può creare problemi. Per ottenere una codifica più compatta si usa spesso la base 16.

In base 16 ci sono, ovviamente, 16 cifre: le cifre decimali da 0 a 9 e poi A,B,C,D,E,F che rappresentano i numeri che in base 10 sono rappresentati come 10, 11, 12, 13, 14, 15. La conversione da e verso la base 2 si ottiene in maniera immediata *impaccando* i bit 4 a 4:

$$[125]_{10} = [01111101]_2 = [7D]_{16}$$

dove $[7]_{16}$ corrisponde ai primi 4 bit $[0111]_2$ e $[D]_{16}$ corrisponde agli ultimi 4 bit $[1101]_2$.

La rappresentazione esadecimale è il modo più consueto di esprimere indirizzi della memoria, che usualmente sono composti di 32 bit e quindi codificati in 8 cifre esadecimali.

Parti frazionarie

Una parte frazionaria è un numero razionale inferiore all'unità.

Come già per gli interi, la conversione della rappresentazione di una parte frazionaria da base decimale a base binaria e viceversa può essere effettuata applicando l'aritmetica della base di partenza o di quella di arrivo.

Nel conto in base di arrivo, la parte frazionaria è sviluppata in forma polinomiale nella base di partenza, poi i singoli termini sono convertiti nella base di arrivo e infine sono eseguite le operazioni in base di arrivo. Ad esempio, dovendo convertire in forma binaria la parte frazionaria decimale $[0.7]_{10}$, l'operazione da eseguire è:

$$0.7 = 7/10 = 0111/1010$$

In questo caso, l'operazione da eseguire in base di arrivo non è un prodotto ma una divisione. Che comunque funziona secondo gli stessi algoritmi che si applicano in base 10. Anche in modo più semplice visto che le singole cifre del quoziente sono 1 o 0 per cui non occorre indovinare quante volte il divisore sta nel dividendo ma serve solo decidere se il divisore nel dividendo ci sta oppure no:

0111.000000		1010
0. 0000	-----	
01110		0.10110
1 1010		----
01000		gli ultimi 4 bits ripetono periodicamente
0 0000		
10000		
1 1010		
01100		
1 1010		
00100		
0 0000		
01000	->periodico	

Si osservi che l'esempio accidentalmente illustra il fatto che non è detto che una parte frazionaria espressa in base 10 ammetta una rappresentazione finita in base 2: $[0.7]_{10} = [0.\overline{10110}]_2$

Nel conto in base di partenza, viene eseguito un algoritmo di moltiplicazioni successive sostanzialmente duale a quello delle divisioni successive: la parte frazionaria viene ripetutamente moltiplicata per due sottraendo ogni volta la parte intera del risultato. La sequenza delle parti intere sottratte fornisce la rappresentazione

ricercata. In questo caso, la natura teorica del procedimento è giustificata dalla osservazione che se $x < 1$ allora esiste unica una successione di valori binari $\{a_k\}_{k=1}^{\infty}$ tali che:

$$x = \sum_{k=1}^{k=\infty} a_k 2^{-k}$$

da cui segue

$$2x = a_1 + \sum_{k=1}^{k=\infty} a_{k+1} 2^{-k}$$

e quindi, essendo $\sum_{k=1}^{k=\infty} 2^{-k} < 1$, segue che $2x > 1$ sse $a_1 = 1$.

1.2.2 Interi senza segno

Un intero senza segno serve a rappresentare una variabile dichiarata in c come istanza del tipo `unsigned int`:

```
unsigned int a;
```

Viene rappresentato su N bit di memoria $\{a_i\}_{i=0}^{N-1}$ che codificano il valore in forma posizionale in base 2:

$$a = \sum_{i=0}^{N-1} a_i * 2^i$$

La rappresentazione codifica tutti e soli i naturali tra 0 a $2^N - 1$ inclusi. Il numero di bits N può variare a seconda dell'architettura della macchina, ma è comunemente uguale a 32. Il che significa che è possibile codificare i numeri da 0 a $2^{32} - 1$. Per figurarsi l'ordine di grandezza della dinamica dei valori si consideri che $2^{10} = 1024 \simeq 10^3$, per cui $2^{32} = 2^{10} * 2^{10} * 2^{10} * 2^2 = 4 * 10^9$.

1.2.3 Caratteri

Un carattere serve a rappresentare una variabile dichiarata in c come istanza del tipo `char`:

```
char c;
```

E' rappresentato in memoria su 8 bit, che codificano un numero intero senza segno tra 0 e 255. La corrispondenza tra i valori numerici dell'intervallo [0,255] e i caratteri è stabilita da una tabella standard, la tabella dei codici ASCII, che include i caratteri alfanumerici (cifre decimali, lettere minuscole e maiuscole), i simboli di interpunzione e le parentesi (, . { etc.), i simboli aritmetici (+,- etc.), altri caratteri di vario genere (@,# etc.), e caratteri di controllo (il segno di a capo, lo spazio di tabulazione, etc.). E' interessante osservare che le cifre decimali sono esse stesse caratteri: il carattere 0 è codificato dal numero 48, il carattere 1 dal 49. Il numero 0 non codifica un carattere utile ed è invece usato come segno di terminazione nella codifica di stringhe.

1.2.4 Interi con segno

Un intero con segno serve a rappresentare una variabile dichiarata in c come istanza del tipo int:

```
int a;
```

Gli interi con segno sono rappresentati su N bit usando una rappresentazione in complemento a 2. In questa codifica, nello sviluppo polinomiale il bit più significativo (Most Significant Bit), ha peso negativo mentre gli altri hanno invece come usuale peso positivo. Dunque, la sequenza di bits $a_{N-1}, a_{N-2}, \dots, a_1, a_0$ codifica il valore:

$$A = -a_{N-1}2^{N-1} + \sum_{n=0}^{k=N-2} a_n 2^n \quad (1.1)$$

I valori rappresentati sono i numeri interi da -2^{N-1} fino a 2^{N-1} , dove il minimo si ottiene asserendo solo il bit MSB (i.e. con la codifica $[100\dots00]_{2c}$) mentre il massimo si ottiene asserendo tutti i bits tranne l'MSB (i.e. con la codifica $[011\dots11]_{2c}$).

Il numero codificato è negativo se e solo se il bit MSB vale 1, il che discende dal fatto che nello sviluppo polinomiale la somma di tutti i termini tolto il primo fa al massimo $\sum_{n=0}^{k=N-2} 2^n = 2^{N-1} - 1$, che non basta a compensare la parte negativa se questa è asserita.

Assegnata la rappresentazione di un numero, il suo opposto (i.e. il numero che ha segno opposto e valore assoluto uguale) può essere determinato complementando i bit uno a uno e poi sommando 1 al risultato. Ad esempio, il numero $A = [110101]_{2c}$ è negativo essendo il suo primo bit asserito; il valore di $-A$ si determina con l'operazione:

```

A=110101
-----
001010+
000001=
-----
001011=-A

```

da cui si deriva agilmente $A = [-11]_{10}$

La rappresentazione in complemento a 2 ha il pregio di unificare il trattamento di somme e sottrazioni e di evitare di distinguere le diverse combinazioni di segno che possono avere i due operandi. Questo semplifica sostanzialmente l'implementazione hardware nella unità aritmetico logica (ALU) di un processore.

In generale, in una operazione di somma, può verificarsi una condizione di *overflow*, ovvero il risultato può eccedere la dinamica dei valori rappresentati. Nel caso di valori codificati come interi senza segno, l'overflow è manifestato dalla presenza di un riporto nella somma dei bit MSB. Nel caso di una codifica in complemento a 2, il riconoscimento dell'overflow è meno diretto: è possibile un overflow solo se i due addendi hanno lo stesso segno (il che è intuitivo in modo immediato); sotto questa condizione, l'overflow si è realizzato se e solo se il risultato della somma è di segno opposto a quello degli operandi (il se è banale, il solo se si dimostra facendo riferimento allo sviluppo in forma polinomiale).

Interessa al programmatore sapere cosa succede in caso di overflow. Nella situazione comune, l'hardware del processore genera un'eccezione, che però è ignorata. L'effetto ultimo è che la somma produce un risultato secondo una aritmetica modulare: sommando 1 al massimo valore positivo, si ottiene il minimo valore negativo.

1.2.5 Valori in virgola mobile

In generale, un numero reale può essere rappresentato in *forma esponenziale* come prodotto:

$$s \cdot m \cdot B^c$$

dove: $s = \pm 1$ è il *segno*; m è un valore reale non negativo detto *mantissa*; c è un intero con segno detto *caratteristica*; B è la base di rappresentazione. Se la base è fissata per convenzione, 2 nel caso della codifica su un calcolatore, il valore del numero è determinato in modo completo dai valori di mantissa, caratteristica e segno.

In generale, uno stesso valore può avere diverse rappresentazioni in forma esponenziale. Ad esempio (in base 10 per la semplicità), il valore 192.5 può essere

rappresentato come $192.5 \cdot 10^0$ ma anche come $19.25 \cdot 10^1$ o come $.1925 \cdot 10^3$. La rappresentazione si dice in *forma normale* quando la mantissa soddisfa il vincolo:

$$1/B \leq m < 1$$

In forma normale, la caratteristica c determina l'ordine di grandezza del valore rappresentato, mentre m è una parte frazionaria che identifica il valore all'interno del suo ordine di grandezza.

La rappresentazione scientifica in forma normale è applicata per rappresentare valori dichiarati in `c` nel tipo `float`:

```
float x;
```

I valori di tipo `float` sono codificati (secondo lo standard IEEE 754) usando 32 bit: 1 bit di segno; 8 bit di caratteristica; 23 bit di mantissa:

s	c	m
1	8	23

Il bit di segno `s` è 0 per i positivi e 1 per i negativi.

La caratteristica è rappresentata in forma polarizzata: piuttosto che rappresentare c come un intero con segno, viene rappresentato il valore $c - 2^7$ che è un intero senza segno che varia tra 0 e 2^8 . Così facendo, ad esempio, la codifica binaria $[10110010]_2$, corrispondente in base 10 al valore $[188]_{10}$, rappresenta la caratteristica $c = [188]_{10} - [128]_{10} = [60]_{10}$. Con 8 bit usati in forma polarizzata, la caratteristica può assumere i valori compresi tra -128 e $+127$. Il che implica che il massimo ordine di grandezza che può essere espresso con una variabile floating point è 2^{127} . Usando ancora una volta la approssimazione $2^{10} \approx 1000$, possiamo provare a immaginare 2^{127} come un 1000^{13} . Si tratta di un ordine di grandezza gigantesco, che sostanzialmente supera il problema dell'overflow incontrato nella codifica dei numeri interi (`int` o `unsigned int`). Si osservi anche che mentre per gli interi il valore assoluto più piccolo ma non nullo è 1, con i `float` è possibile rappresentare valori nell'ordine di 2^{-128} .

La mantissa è una parte frazionaria (ovvero un valore non negativo minore di 1):

$$m = \sum_{k=1}^{\infty} m_k 2^{-k}$$

Essendo il valore in forma normale risulta $\frac{1}{2} \leq m < 1$, da cui segue che il bit più significativo m_1 è uguale a 1; se infatti la prima cifra fosse 0, il valore di m risulterebbe minore secco di $\frac{1}{2}$ essendo per qualunque valore di N

$$\sum_{k=2}^N 2^{-k} < \frac{1}{2}$$

Grazie a questa condizione, la codifica del bit m_1 può essere omessa, e i 23 bit disponibili per la rappresentazione della mantissa sono usati per codificare i bit da m_2 a m_{24} .

$$m = 2^{-1} + \sum_2^{24} m_k 2^{-k}$$

Il fatto di disporre di un numero finito di bits per la codifica fa sì che la mantissa possa rappresentare solo in sottoinsieme finito dei valori razionali compresi tra $\frac{1}{2}$ e 1. Questo induce un problema di *precisione finita*, ovvero un errore di approssimazione nella rappresentazione di numero razionale attraverso un valore float.

Con 23 bits dedicati a rappresentare la mantissa, per ogni ordine di grandezza identificato dalla caratteristica è possibile rappresentare 2^{23} valori distinti. Il fatto di avere lo stesso numero di valori codificati su diversi ordini di grandezza indica che la densità con cui sono coperti i razionali non è omogenea e quindi non è omogenea neppure la distanza che può correre tra un valore razionale e la sua migliore approssimazione nell'insieme dei float. Conviene allora che esprimiamo l'errore di precisione finita in forma di errore relativo. Se x è un valore razionale e \bar{x} è la sua approssimazione per difetto nell'insieme dei float, l'errore relativo è

$$\epsilon_r = \frac{x - \bar{x}}{x} \quad (1.2)$$

Su tale errore è possibile derivare una stima che è invariante rispetto all'ordine di grandezza di x :

$$\epsilon_r = \frac{x}{x - \bar{x}} = \frac{sm2^c - s\bar{m}2^c}{sm2^c} = \frac{m - \bar{m}}{m} \quad (1.3)$$

dove m e \bar{m} denotano la mantissa della rappresentazione esatta e di quella approssimata di x e \bar{x} , rispettivamente. In generale, m può essere rappresentato come parte frazionaria con un numero non finito di termini

$$m = 2^{-1} + \sum_{k=2}^{\infty} m_k 2^{-k} \quad (1.4)$$

mentre la sua approssimazione per difetto nell'insieme dei float è ottenuta troncando lo sviluppo al 24-mo bit:

$$\bar{m} = 2^{-1} + \sum_{k=2}^{24} m_k 2^{-k} \quad (1.5)$$

Sostituendo in Eq.(1.3) otteniamo allora:

$$\epsilon_r = \frac{m - \bar{m}}{m} = \frac{\sum_{k=25}^{\infty} m_k 2^{-k}}{2^{-1} + \sum_{k=2}^{\infty} m_k 2^{-k}} \quad (1.6)$$

Il denominatore è evidentemente maggiore di -1 , mentre il numeratore è minore di 2^{-24} essendo:

$$\sum_{k=25}^{\infty} m_k 2^{-k} \leq \sum_{k=25}^{\infty} 2^{-k} = 2^{-24} \sum_{k=1}^{\infty} 2^{-k} = 2^{-24} \quad (1.7)$$

Sostituendo in Eq.(1.6) otteniamo infine:

$$\epsilon_r \leq \frac{2^{-24}}{\frac{1}{2}} = 2^{-23} \quad (1.8)$$

Ancora una volta, per avere un senso pratico della dimensione di 2^{-23} , possiamo approssimare $2^{10} \simeq 1000$ da cui segue $2^{-23} \simeq \frac{1}{8 \cdot 10^6} \simeq 10^{-7}$.

L'errore assoluto è il prodotto tra l'ordine di grandezza e l'errore relativo:

$$\epsilon_a = x * \frac{x - \bar{x}}{x} \leq x * 2^{-23} \leq 2^c * 2^{-23}$$

Si osservi che se x cade nell'ordine di grandezza compreso tra $1/2$ e 1 allora $2^c = 1$ e l'errore assoluto coincide con quello relativo; se invece c è nell'ordine di grandezza massimo, l'errore assoluto diventa una enormità: $2^{127} * 2^{-23} = 2^{104}$. Nella pratica questo non è un problema perché nell'ingegneria gli errori contano in relativo.

L'impatto dell'errore di rappresentazione può essere ridotto usando valori in precisione doppia che in C sono realizzati con il tipo double:

```
double y;
```

I valori di tipo double sono codificati su 64 bits: 1 di segno, 8 di caratteristica e 23+32 di mantissa. In tal modo l'errore relativo si riduce a 2^{-55} .

Errori di aritmetica in precisione finita Il problema della precisione finita non si limita al fatto che un valore razionale debba essere approssimato nel momento in cui viene codificato attraverso un float. Succede anche che una operazione aritmetica su due valori float in generale restituisce un valore che non è codificabile a sua volta come float richiedendo una ulteriore approssimazione. In sostanza, l'insieme dei valori del tipo float non è chiuso rispetto a operazioni aritmetiche elementari quali la somma o il prodotto.

Per comprendere il problema è utile seguire il modo con cui nella unità aritmetico logica (ALU) di un processore viene eseguita l'operazione di somma di due valori float: le caratteristiche dei due addendi sono allineate al valore più alto facendo scorrere la caratteristica dell'addendo minore; la caratteristica comune viene portata a fattore e le due mantisse sono sommate; se necessario la caratteristica viene nuovamente fatta scorrere fino a riportare la mantissa in forma normale.

Esemplifichiamo usando per semplicità una rappresentazione decimale con tre cifre di mantissa:

$$.465 * 10^1 + .997 * 10^3 = (.00465 + .997) * 10^3 = 1.00165 * 10^3 = .100165 * 10^4$$

Si osservi che a seguito della prima operazione di allineamento della caratteristica la rappresentazione esatta della mantissa dell'addendo minore richiede un numero aumentato di cifre, e comunque il numero di cifre della mantissa del risultato in generale eccede il formato della rappresentazione. Mentre il primo problema viene gestito attraverso una rappresentazione estesa degli operandi all'interno della ALU, la necessità di riportare il risultato nel formato di rappresentazione esterno comporta il troncamento di un numero di cifre che nel caso pessimo sono pari alla differenza tra le caratteristiche degli addendi:

$$.465 * 10^1 + .997 * 10^3 = .100165 * 10^4 \rightarrow 0.100 * 10^4$$

Si osservi che nel caso in cui la differenza tra le caratteristiche dei due addendi eccede il numero di cifre nella rappresentazione della mantissa, il risultato diventa esattamente uguale all'addendo maggiore, ignorando del tutto il valore del minore.

Come ultima osservazione, vale la pena di rimarcare che la presenza diffusa di errori di precisione finita rende estremamente poco robusto il test di uguaglianza su valori float. Per questo, il test di uguaglianza è usualmente rilassato nella forma di un test sulla distanza tra i due valori in riferimento alla precisione di macchina.

Interi e float

E' utile soffermarsi a riflettere sul perché in un linguaggio di programmazione esistano due tipi numerici `int` e `float` visto che nella teoria dei numeri gli interi sono un sottoinsieme dei razionali.

In effetti gli `int` della programmazione non sono gli interi dell'aritmetica ma sono tutti e soli quelli che cadono entro un range, tipicamente tra -2^{31} e $+2^{31}-1$. Analogamente, i `float` non sono i razionali ma un loro sottoinsieme finito formato da tutti e soli i valori razionali che cadono entro la dinamica della rappresentazione e che ammettono una forma normalizzata con una mantissa nella quale siano nulli tutti i bit oltre il 24 esimo. E vale la pena di osservare che gli `int` non sono neppure un sottoinsieme dei `float`, visto che qualunque numero codificato in `int` con più di 25 bit non può essere rappresentato esattamente come `float`.

Oltre alla differenza nell'insieme dei valori rappresentati, la differenza sostanziale tra i tipi `int` e `float` attiene anche alla semantica effettiva associata alle operazioni aritmetiche, che è soggetta a limiti diversi dovuti alla finitezza della rappresentazione. Mentre gli `int` sono soggetti al problema dell'overflow ma garantiscono altrimenti una aritmetica esatta, i `float` non incorrono in errori di overflow, ma introducono errori di approssimazione in maniera diffusa.

1.3 Rappresentazione delle istruzioni

Un programma è usualmente scritto usando un linguaggio di alto livello, il c nel riferimento di questo testo. Per essere eseguita, la codifica di alto livello deve però essere tradotta in una sequenza di istruzioni di basso livello che possano essere tradotte in forma numerica, caricate nella memoria di un elaboratore ed eseguite da un processore.

1.3.1 Assembler

L'assembler è il linguaggio delle istruzioni che possono essere eseguite su un processore. Ogni processore ha il suo assembler e i due sono mutuamente connessi, esistendo una corrispondenza stretta tra l'organizzazione hardware del processore e la struttura delle istruzioni che esso può eseguire. Questa mutua dipendenza è bene espressa dicendo che il processore e il suo assembler costituiscono un *sistema di calcolo*.

In generale, sistemi di calcolo diversi si distinguono per dettagli che qualificano aspetti come la ricchezza di istruzioni del linguaggio, l'efficienza di esecuzione delle sue istruzioni, la loro regolarità, la complessità tecnologica dell'hardware, la economicità di realizzazione del processore, la sua capacità di essere integrato nella organizzazione complessiva di un elaboratore.

I diversi sistemi sono però equivalenti per quanto attiene alla potenza espressiva: qualsiasi due processori che abbiano un qualche senso pratico sono in grado di eseguire tutti e soli gli stessi programmi. Per questo la teoria della computabilità è convenientemente basata su un sistema di calcolo astratto, la macchina di Turing, che qualifica la potenza espressiva di un sistema di calcolo senza fare riferimento all'implementazione di uno specifico processore. Il valore di questa astrazione è sostenuto dalla cosiddetta Tesi di Church, la quale postula che non sia possibile realizzare un sistema di calcolo capace di eseguire programmi che non possano essere eseguiti sulla macchina di Turing. Un sistema di calcolo che realizza la potenza espressiva della macchina di Turing si dice completo.

Negli obiettivi della nostra trattazione interessa intuire concretamente in che modo un programma c possa essere tradotto in forma eseguibile. Per questo, piuttosto che la macchina di Turing preferiamo considerare un sistema di calcolo concreto. In particolare consideriamo un frammento di un assembler di tipo RISC (Reduced Instruction Set Computer), il MIPS, che gira(va) sui processori R4000 della Silicon Graphics alla fine degli anni 90. È inutile dire che si tratta di un sistema di calcolo completo, e quindi equivalente a quello di qualsiasi altro processore.

Calcolo delle espressioni e esecuzione di istruzioni semplici

Per tradurre le espressioni del linguaggio c occorre innanzitutto qualcosa che realizzzi le operazioni aritmetiche e logiche e gli assegnamenti. L'istruzione

```
add a, b, c;
```

calcola la somma di b con c e scrive il risultato in a. In modo analogo operano un numero di altre istruzioni come sub, mult, and, che realizzano i diversi operatori aritmetici e logici. È rilevante osservare che add può realizzare un assegnamento quando uno dei due addendi vale 0: l'istruzione assembler add a,b,0; assegna il valore di b alla variabile a. L'istruzione add ha sempre due addendi e la somma di più addendi si ottiene per associazione. Ad esempio, l'istruzione c:

```
x=a+b+c;
```

viene tradotta in due istruzioni assembler

```
add tmp, a, b;  
add x, tmp, c;
```

È rilevante osservare fino da questo caso elementare come tra istruzioni c e assembler esista un rapporto uno-a-molti: una singola istruzione del c corrisponde a una o più istruzione dell'assembler. Nella traduzione sono anche introdotte delle variabili temporanee aggiuntive che non compaiono nel programma c, e viene determinata una particolare sequenzializzazione delle operazioni di somma dei tre addendi. La traduzione e le scelte che essa comporta sono responsabilità del compilatore che esegue la traduzione da c ad assembler.

Per ragioni tecnologiche, le operazioni aritmetiche e logiche non sono eseguite su variabili in memoria ma su variabili contenute in un banco di registri contenuti all'interno della CPU che permettono un accesso più veloce. Nel caso dell'R4000 i registri sono 32 e sono denotati come \$0, \$1, ... \$31. I registri sono indistinti, salvo alcune funzionalità particolari sulle quali non ci serve soffermarci. Segnaliamo solo che all'inizio di ogni istruzione \$0 contiene il valore 0.

Dunque l'istruzione add si scrive in realtà come:

```
add $x, $y, $z;
```

e l'espressione precedente si traduce nella forma

```
add $5, $2, $3;  
add $6, $5, $4;
```

dove la variabile temporanea è stata realizzata sul registro \$5, dove si è assunto che le variabili a, b e c siano memorizzate nei registri \$2, \$3 e \$4, e dove il risultato finale è calcolato nel registro \$6. Si osservi che la scelta dei registri sui quali caricare le variabili e memorizzare i passaggi intermedi non è univoca ed è ancora a carico del compilatore.

Avendo limitato le operazioni aritmetiche e logiche a operandi memorizzati nei registri, diventa necessario disporre di istruzioni per trasferire i dati dalla memoria ai registri. Facendo ancora riferimento all'esempio precedente, occorre caricare a, b e c in \$2, \$3 e \$4, e occorre poi scaricare \$6 in d.

L'istruzione load-word

`lw $x, Base[$y]`

carica in \$x il contenuto della parola (4 byte) collocata in memoria a partire dall'indirizzo determinato dalla somma del numero Base con il valore contenuto nel registro \$y. Viceversa, l'istruzione store-word scrive il contenuto del registro \$x nella locazione di memoria all'indirizzo determinato come risultato della somma del numero Base con il valore contenuto in \$y:

`sw $x, Base[$y]`

È interessante osservare come le istruzioni di trasferimento tra memoria e registri riflettano un orientamento al trattamento di sequenze di variabili dello stesso tipo, che nel caso particolare del c sono realizzate come arrays: incrementando il contenuto del registro indice \$y è possibile indirizzare una sequenza di variabili contigue. Il fatto che questo sia supportato fino dal livello dell'assembler suggerisce una riflessione sul fatto che il trattamento di arrays costituisce un meccanismo fondamentale della programmazione: esso permette di applicare una stessa istruzione ad una molteplicità di variabili diverse.

Poiché un'espressione c può contenere anche costanti (e.g. $d=a-b+15$), è necessario disporre nel linguaggio assembler di istruzioni capaci di trattare operandi costanti. Questo è realizzato con istruzioni aritmetiche con operandi il cui valore non è contenuto in un registro ma è invece rappresentato in modo *immediato* nell'istruzione. Facendo ancora riferimento all'operazione di somma, l'istruzione

`addi $x, $y, value`

scrive nel registro \$x il risultato della somma del valore value con il valore contenuto nel registro \$y.

È a questo punto possibile riportare il codice assembler che traduce l'istruzione $a=b+15-c[2]-c[3] ;$:

```

# traduzione asm dell'istruzione c:    a=b+15-c[2]-c[3];
# il valore dell'espressione \`e calcolato in $5
# l'indice sull'array c \`e mantenuto in $6
# i valori dell'array c sono caricati in $7
# supponiamo che a, b, c siano agli indirizzi 100, 104, e 108
# NB: il registro $0 contiene il valore 0

lw    $5,104[$0];
addi $5,$0,15;
lw    $6,$0,2;
lw    $7,108[$6];
sub  $5,$5,$7;
addi $6,$6,1;
lw    $7,108[$6];
sub  $5,$5,$7;
sw    $5,100[$0];

```

Controllo del flusso

Con le istruzioni aritmetico-logiche e quelle di trasferimento tra registri e memoria è possibile scrivere un programma che esegue sempre la stessa sequenza di istruzioni, ovvero un programma nel quale ciascuna istruzione trasferisce il controllo alla istruzione che la segue immediatamente. Per rendere il flusso di controllo ramificato e dipendente dai dati occorre qualcosa che permetta di selezionare la prossima istruzione in base all'esito di un test. Questa capacità è fornita in assembler dall'istruzione di conditional branch:

```
beq $x,$y,label;
```

Nella esecuzione della istruzione, vengono comparati i contenuti dei registri \$x e \$y; se il contenuto di \$x è uguale al contenuto di \$y, il controllo viene passato all'istruzione immediatamente successiva all'etichetta label, altrimenti il controllo procede sequenzialmente alla istruzione successiva. Questo corrisponde sostanzialmente all'istruzione condizionale if del linguaggio c, salvo che in questo caso l'espressione su cui viene deciso il salto ha la forma minimale di un test di uguaglianza tra i contenuti di due registri.

Per realizzare test basati non sull'uguaglianza di valori ma sul soddisfacimento di disequazioni, l'istruzione slt setta il registro \$x al valore 1 se \$y>\$z e al valore 0 se \$y<=\$z:

```
slt $x,$y,$z;
```

L'uso combiato di `beq` e `slt` permette di eseguire i diversi test relazionali `>`, `>=`, `<`, `<=`.

Esiste infine l'istruzione di salto incondizionato `jump`:

```
j label;
```

la quale trasferisce il controllo all'istruzione immediatamente successiva all'etichetta `label`. Il salto incondizionato, che corrisponde al `goto` del linguaggio c, non aggiunge potenza espressiva rispetto a quella già conferita dalle istruzioni `beq` e `slt`. È un elemento di convenienza dettato da ragioni tecnologiche, legate al formato numerico con cui sono codificate le istruzioni assembler.

In un assembler di tipo RISC non esistono istruzioni che traducono direttamente i costrutti di iterazione `for`, `while` e `do while` del linguaggio c. Queste in effetti possono essere realizzate con una opportuna combinazione di salti condizionati, come del resto è possibile intuire considerando la rappresentazione in flow charts in Fig.1.3.

Esempio

Consideriamo come esempio un frammento di codice c che calcola la somma dei valori su un vettore `V` usando il costrutto di iterazione `for`:

```
/* codice c che calcola la somma dei valori di un vettore */
int V[128];
int sum;
int count;
sum=0;
for(count=0;count<128; count++)
    sum=sum+V[count];
```

In maniera verosimile, il frammento viene compilato in assembler nella forma:

```
# l'array V e' allocato a partire dall'indirizzo 100 incluso
# e occupa fino all'indirizzo 612 escluso
# sum e' all'indirizzo 612
# count non e' mai usata
# la somma e' accumulata sul registro $4
# l'indice count e' mantenuto in $5
# la costante 128 e' caricata in $6
# i valori dell'array sono caricati in $7
```

```

# Nota: slt $1,$2,$3 setta $1 al valore 1 se $2<$3
# e al valore 0 altrimenti
#
        addi $4,$0,0;
        addi $5,$0,0;
        addi $6,$0,128;
start_loop:slt $1,$5,$6;
            beq $0,$1,end_loop;
            lw $7,100[$5];
            add $4,$4,$7;
            addi $5,$5,1;
            j start_loop;
end_loop: sw $4,612[$0];

```

1.3.2 Linguaggio macchina

L'assembler, come il c, usa simboli per denotare registri, operatori e etichette. Ad esempio si scrive add per denotare una operazione di somma, \$5 per indicare un registro, end_loop: per denotare la posizione di una istruzione. Per potere essere caricato in memoria e poi eseguito su un processore, un codice assembler deve essere tradotto ulteriormente in una forma numerica. Questa costituisce il cosiddetto *linguaggio macchina* ed è generato dall'*assemblatore*.

Mentre il lavoro svolto dal compilatore nella traduzione da c ad assembler è una operazione complessa che include scelte e quindi dà spazio ad un problema di ottimizzazione, la traduzione dall'assembler al linguaggio macchina è una operazione univoca e tutto sommato banale. Ci interessa comunque considerarla perché essa rende esplicite le ragioni di alcune caratteristiche dell'assembler, e perché permette di intuire come un programma c possa diventare una sequenza di numeri binari.

Nel MIPS, tutte le istruzioni sono codificate su 32 bits. I primi 6 bits formano il *campo operativo* che codifica il tipo di istruzione, permettendo un linguaggio di $2^6 = 64$ istruzioni diverse. I successivi 26 bits sono organizzati diversamente a seconda dell'informazione che caratterizza le diverse istruzioni.

add e addI

Le istruzioni add e sub sono codificate su 6 campi, di 6, 5, 5, 5, 5 e 6 bits rispettivamente:

```
add $rd, $rs, $rt
```

op=0	rs	rt	rd	-	funct=32
6	5	5	5	5	6

```
sub $rd, $rs, $rt
```

op=0	rs	rt	rd	-	funct=34
6	5	5	5	5	6

Il campo **op** vale 0 per tutte le istruzioni aritmetico logiche, che sono differenziate nel campo **funct**. Ad esempio, **funct** vale 32 per la somma e 34 per la differenza.

I campi **rs**, **rt** e **rd** codificano i tre registri interessati dall'operazione. Essendovi 32 registri, ciascuno di essi è identificato in modo univoco con $\ln_2(32) = 5$ bits che esprimono un numero tra 0 e 31. È interessante osservare che il registro destinazione **\$rd** è codificato in terza posizione, pur essendo il primo operando nella rappresentazione simbolica dell'assembler; la ragione dell'inversione ha a che fare con il modo con cui le istruzioni sono trattate nel processore ed è un esempio emblematico di come siano intimamente legate la definizione del linguaggio assembler e la organizzazione hardware del processore.

Il campo di 5 bit che precede il campo **funct** è irrilevante nel caso della somma e della differenza e serve solo per particolari istruzioni di shift dei bit. La sua presenza è un buon esempio di come l'approccio RISC privilegi la regolarità nel formato delle istruzioni rispetto ad una possibile ottimizzazione della loro dimensione.

Il formato dell'istruzione **add** è chiamato *formato R*, come Register, perché tutti gli operandi coinvolti sono definiti mediante il numero del registro che li contiene.

L'istruzione **addI** richiede la codifica di due soli registri e di un campo immediato per la rappresentazione della costante, oltre ovviamente al campo **op** che in questo caso vale 8. Il formato che ne risulta è detto *formato I* come Immediate:

```
addi $rd, $rs, value
```

op=8	rs	rd	value
6	5	5	16

La codifica di **addi** rende evidente il modo in cui le costanti sono codificate come parte delle istruzioni, e dunque separate rispetto alla regione di memoria che ospita le variabili. Si può osservare come la costante **value** sia codificata su 16 bits, che

permettono la rappresentazione di 65536 valori. Oltre questo range, il compilatore assume altri accorgimenti, che non ci interessa approfondire.

load word e store word

Le istruzioni load word (**lw**) e store word (**sw**) sono ancora codificate in formato I e condividono la stessa codifica di **addI**, salvo il diverso valore del campo **op** (35 per **lw**, 43 per **sw**):

lw \$rd,base[\$rt]

op=35	rt	rd	base
6	5	5	16

sw \$rs,base[\$rt]

op=43	rt	rs	base
6	5	5	16

Si osservi che il terzo campo della codifica (i bits 11:15) hanno diverso significato: nell'istruzione **lw**, il campo **rd** codifica l'indirizzo del registro sul quale viene scritto il valore prelevato dalla memoria; nell'istruzione **sw**, il campo **rs** codifica invece l'indirizzo del registro dal quale è letto il valore che viene scritto in memoria.

branch equal, set on less e jump

L'istruzione branch equal **beq** codifica i due registri comparati e la distanza dalla istruzione corrente alla quale si trova l'istruzione a cui trasferire il controllo nel caso in cui venga effettuato il salto. L'informazione è quindi ancora codificata in formato I, con valore del campo **op** uguale a 4:

beq \$rs, \$rt, label

op=4	rs	rt	label
6	5	5	16

L'istruzione set-on-less **slt**, codifica tre registri ed è quindi rappresentata in formato R con campo **op** uguale a 0 (in effetti è una istruzione logico aritmetica) e campo **funct** uguale a 42:

`slt $rd, $rs, $rt`

op=0	rs	rt	rd	-	funct=42
6	5	5	5	5	6

È utile osservare che la disponibilità di soli 16 bit nel campo immediato della istruzione `beq` permette di indirizzare un massimo di 65536 locazioni distinte. Limitando le possibili destinazioni ai soli multipli di 4, questo permette di saltare entro un range di ± 128 Kbytes. Questi sono sicuramente sufficienti per salti locali, ma possono non bastare per trasferire il controllo nella chiamata di una funzione. Il limite è superato attraverso l'uso del branch equal in combinazione con il jump incondizionato `j label`; (e in casi estremi con la ulteriore istruzione di salto `jr $x`).

L'istruzione di salto incondizionato `J label` è codificata in *formato J*, con campo `op` uguale a 2 e con i rimanenti 26 bit destinati alla rappresentazione dell'ampiezza del salto:

`j label`

op=2	label
6	26

Esempio

Possiamo a questo punto riprendere il caso del frammento di codice che calcola la somma dei valori su un vettore per il quale nel capitolo 1.3.1 avevamo esplicitato la compilazione da c ad assembler, e vedere come a sua volta la rappresentazione assembler viene codificata in forma numerica:

```

        addi $4,$0,0;
        addi $5,$0,0;
        addi $6,$0,128;
start_loop: slt $1,$5,$6;
            beq $0,$1,end_loop;
            lw $7,100[$5];
            add $4,$4,$7;
            addi $5,$5,1;
            j start_loop;
end_loop:  sw $4,612[$0];

```

8	0	4	0			
8	0	5	0			
8	0	6	128			
0	5	6	1	-	42	
4	0	1	+16			
35	5	7	100			
0	4	7	4	-	32	
8	5	5	1			
2	-20					
43	0	4	612			

L'esempio, tutto sommato non aggiunge nulla a quanto detto fino qui, salvo l'impatto visivo di come una sequenza di istruzioni simboliche diventa una sequenza di numeri. I numeri sono mostrati in forma decimale, ma è chiaro che poi nella concreta realizzazione sono rappresentati in formato binario.

Come unica osservazione vale la pena notare che, mentre la forma dell'assembler suggerisce che la label di un `j` o un `beq` denotino indirizzi assoluti, in realtà esse codificano lo spiazzamento tra l'istruzione corrente e quella a cui deve essere passato il controllo. Il calcolo dello spiazzamento è eseguito dall'assemblatore come parte della traduzione in linguaggio macchina, con la fondamentale semplificazione che tutte le istruzioni occupano 4 bytes (almeno nel processore RISC che abbiamo considerato).

1.3.3 Esecuzione su un processore

Una volta che abbiamo compreso come un programma c possa essere compilato in assembler e poi codificato nella forma numerica del linguaggio macchina, diventa possibile comprendere il principio di funzionamento secondo il quale un processore (Central Processing Unit - CPU) può eseguire un programma caricato in memoria.

Per questo presentiamo lo schema di un processore capace di eseguire le istruzioni `add`, `addI`, `lw`, `sw` e `beq` del frammento di assembler illustrato nel Capitolo 1.3. Lo schema può essere facilmente esteso per trattare anche l'istruzione `j`, omessa per semplicità.

Si tratta di un modello, che non rappresenta la realtà del processore RISC4000 dal quale abbiamo preso avvio in questa trattazione e che omette di evidenziare le molte possibili ottimizzazioni che fanno l'essenza del progetto e dello studio dei calcolatori elettronici. Il modello ha però la capacità di completare la concretizzazione del processo che conduce dalla rappresentazione in un linguaggio di alto livello alla esecuzione su un processore, evidenziando anche lo stretto legame che esiste tra un assembler e il suo processore.

Componenti della CPU

Lo schema di Fig.1.4 contiene cinque *blocchi funzionali* fondamentali: il registro *program counter* (PC), la *memoria del programma*, il *banco dei registri*, la *memoria dati*, la *unità aritmetico logica* (ALU), la *unità di controllo* (CU). Ci sono poi alcuni componenti minori: alcuni *multiplexor* (rappresentati con la forma di rettangoli verticali arrotondati), un *decoder*, un *modulo di estensione* da 16 a 32 bit, due *sommatori*, una *porta AND*.

I blocchi sono collegati attraverso linee che connettono porte per l'ingresso e l'uscita di dati, per l'indirizzamento e il controllo (le linee di controllo sono rappresentate a tratteggio). Le linee possono trasferire segnali di un solo bit o

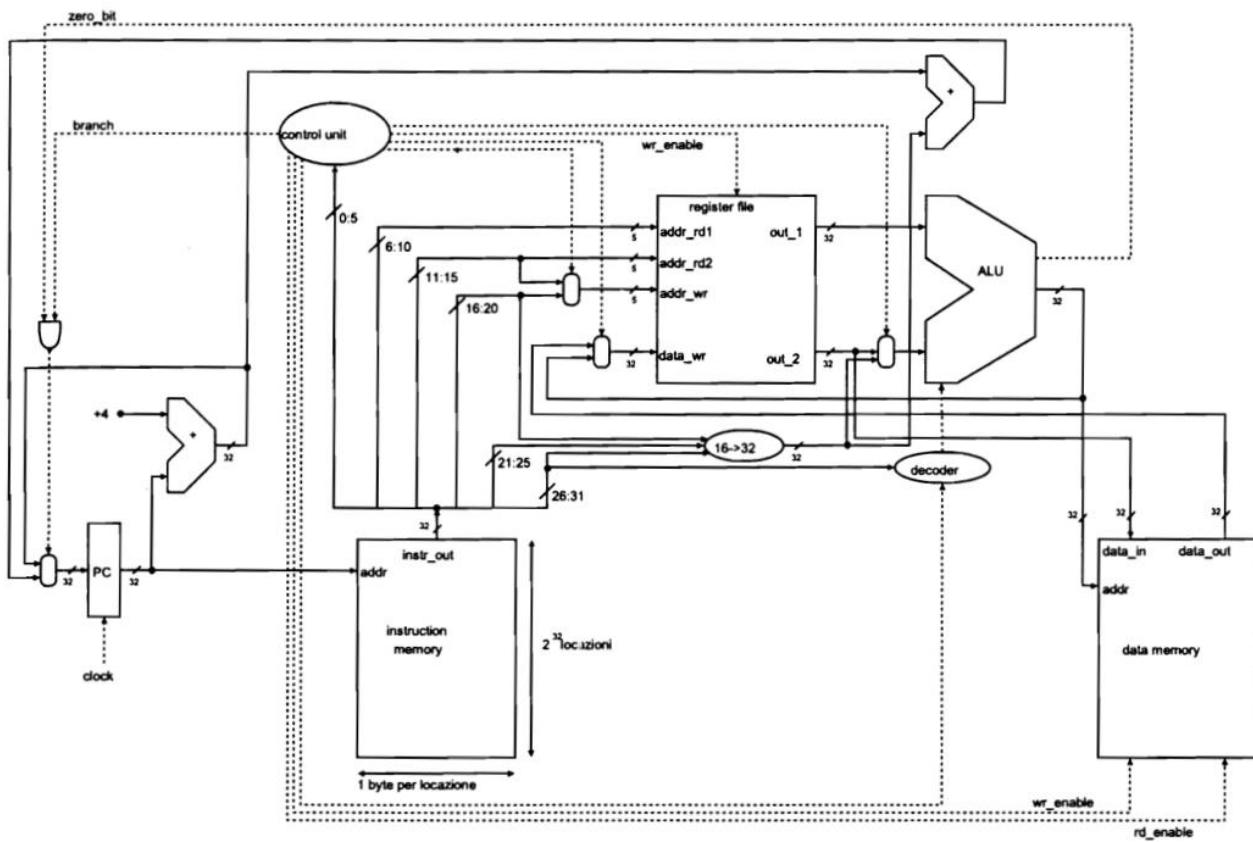


Figura 1.4: Lo schema di una CPU in grado di eseguire le istruzioni `add`, `sw`, `lw`, `beq`, `addI`.

di più bits paralleli. In quest'ultimo caso viene spesso indicata la dimensionalità della linea. Si osservi che gli incroci nel disegno non sono punti di contatto salvo il caso in cui siano marcati esplicitamente con un punto. Questo avviene laddove il segnale di una linea viene replicato su due linee con diverse destinazioni.

- Una *memoria* è un dispositivo che contiene una sequenza lineare di locazioni da un byte (8 bit) che possono essere lette e scritte nel corso della computazione. Da un punto di vista funzionale una memoria è caratterizzata da un ingresso di indirizzamento (*addr*), un ingresso e una uscita dati (*data_in* e *data_out*), due segnali di controllo (*read_enable* e *write_enable*).

L'ingresso di indirizzamento *addr* seleziona una locazione nella memoria. Il numero di bits che formano l'indirizzo determina il numero di locazioni indirizzabili e quindi la ampiezza della memoria: con indirizzi a 32 bits la memoria può contenere un massimo di 2^{32} locazioni.

Gli ingressi di controllo *read_enable* e *write_enable* (un bit ciascuno) controllano le operazioni di lettura dalla memoria e di scrittura sulla memoria.

Quando il bit *read_enable* viene asserito, la memoria risponde presentando sull'uscita *data_out* il contenuto della locazione di memoria selezionata dall'indirizzo *addr*. In realtà per ragioni di efficienza e semplicità la lettura produce in uscita una parola costituita dai 4 bytes che seguono all'indirizzo selezionato da *addr*.

Viceversa, quando viene asserito il bit *write_enable*, la memoria risponde scrivendo a partire dalla locazione all'indirizzo *addr* la parola presentata all'ingresso *data_in*. È utile osservare che in assenza di un segnale di *write_enable*, il valore presentato sulla porta di ingresso è irrilevante.

Si osservi che per la memoria delle istruzioni i due segnali di controllo sono omessi, sottendendo che nel funzionamento illustrato assumiamo che la memoria abbia il segnale di *read_enable* asserito con continuità.

Nello schema di Fig.1.4 figurano due blocchi di memoria distinti nei quali si assume siano caricate le istruzioni del programma e siano memorizzati i dati su cui esso opera nel corso della computazione. I due blocchi rappresentano in realtà due moduli che interfacciano la CPU con la memoria, che si trova invece all'esterno della CPU, che è unica per dati e istruzioni e che ha tipicamente una dimensione inferiore a quella massima indirizzabile. Come parte dell'interfacciamento i due blocchi provvedono a realizzare funzioni di caching e prefetch, che sono decisive ai fini della prestazione di un processore ma che non sono essenziali ai fini della nostra trattazione.

- Il *program counter* (PC) è un registro a 32 bit dedicato a contenere l'indirizzo della istruzione corrente. Quando viene asserito il segnale di controllo, il registro memorizza il valore presentato sulla porta di ingresso e lo mantiene con continuità sulla porta di uscita fino al successivo aggiornamento. Nello schema di Fig.1.4 il segnale di controllo è agganciato al clock di sistema, per cui ad ogni colpo di clock il PC aggiorna il suo valore determinando l'esecuzione di una nuova istruzione.
- Il *banco dei registri* è un dispositivo che contiene al suo interno 32 registri di 32 bits ciascuno, concettualmente equivalenti al registro program counter. Il banco ha però la particolarità di essere organizzato in maniera tale da permettere contemporaneamente la lettura di due registri e la scrittura di un terzo (eventualmente con due o anche tre registri coincidenti). Il che serve ad esempio nella esecuzione dell'istruzione add.

Il banco ha tre ingressi di indirizzamento che selezionano due registri da cui leggere (*addr_rd1* e *addr_rd2*) e un registro su cui scrivere *addr_wr*. Poiché il banco contiene 32 registri, ciascun indirizzo è composto di 5 bits ($5 = \ln_2(32)$).

Le due uscite dati `out_1` e `out_2` presentano il contenuto dei registri indirizzati da `addr_rd1` e `addr_rd2` rispettivamente. Le due uscite hanno la dimensione di 32 bits ciascuna, essendo 32 il numero di bits in ciascuno dei registri nel banco.

Il segnale di controllo `wr_enable` abilita la scrittura del dato presentato sull'ingresso `data_wr` sul registro indirizzato da `addr_wr`. In assenza del segnale, il valore presentato all'ingresso di scrittura è irrilevante.

- La *Arithmetic Logic Unit* (ALU) è un dispositivo capace di eseguire operazioni aritmetiche e logiche. Ha due ingressi per gli operandi e una uscita per il risultato che hanno una stessa dimensione (32 bit nella architettura considerata), un ingresso di controllo di qualche bit che seleziona l'operazione che deve essere eseguita, e una uscita di controllo di un bit, lo *zero bit*, che viene asserito quando l'operazione eseguita nella unità restituisce valore zero.
- Il *sommatore* è concettualmente analogo a una ALU ma è semplificato per eseguire la sola operazione di addizione.
- Il *multiplexor* è un selettore di canale. È un dispositivo a due ingressi, una uscita e un segnale di controllo: in base al valore del segnale di controllo, il multiplexor presenta in uscita uno dei due ingressi. Il valore dell'altro ingresso rimane irrilevante.
- La *control unit* (CU) è una rete combinatoria. Riceve in ingresso un segnale di 6 bits che codifica il campo operativo `op` della istruzione in esecuzione e genera in uscita una varietà di segnali di controllo che pilotano il funzionamento dei componenti della CPU. In sostanza, la control unit configura la CPU in modo da adattarla alla particolare istruzione in esecuzione.
- La porta *AND* ha due ingressi da un bit ciascuno e genera una uscita di un bit che viene asserito se e solo se entrambi gli ingressi hanno il valore 1.
- Il *decoder* associato alla ALU è una rete combinatoria che genera il segnale di controllo che pilota la particolare operazione aritmetico logica eseguita. Nel caso di una istruzione aritmetico logica (e.g. l'istruzione `add`), il segnale di controllo per la ALU è derivato dal campo `func` codificato negli ultimi 6 bits dell'istruzione. Per tutte le altre istruzioni (e.g. le istruzioni `sw`, `lw`, `beq`, `addI`), il segnale di controllo per la ALU è derivato direttamente dai segnali di controllo generati dalla control unit.
- Il modulo di *estensione da 16 a 32 bits* estende il numero di bits con cui è rappresentato un campo immediato (codificato sugli ultimi 16 bits dell'istruzione) in modo da renderlo compatibile con l'ingresso della ALU e di altri

blocchi che operano su segnali 32 bits. Concettualmente l'estensione è analoga alla operazione con cui si estende il numero di cifre che codificano un valore decimale aggiungendo degli zeri in testa (e.g. 260165 → 00260165). Nella pratica l'operazione è appena più complessa per il fatto che i dati sono codificati in complemento a due.

Esecuzione delle istruzioni

add: Consideriamo il caso in cui il PC contenga l'indirizzo di una locazione di memoria in cui è codificata una istruzione `add $rd,$rs,$rt`. In questa condizione la memoria delle istruzioni presenta in uscita una istruzione nel formato descritto nel capitolo 1.3.2:

op=0	rs	rt	rd	-	funct=34
0:5	6:10	11:15	16:20	21:25	26:31

In base ai collegamenti dello schema di Fig.1.4, i bit 0:5 che codificano il campo op sono presentati in ingresso alla logica di controllo, la quale li decodifica per riconoscere l'istruzione in esecuzione e quindi per generare i segnali di controllo che configurano la CPU per eseguire l'istruzione: i segnali di selezione per i 4 multiplexors, il segnale `write_enable` per il banco dei registri, e il segnale che configura il decoder in modo che esso generi un segnale di comando alla ALU in base al valore dell'ingresso funct.

Fig.1.5 rappresenta la configurazione risultante escludendo i blocchi funzionali e i collegamenti che non sono attivati dai segnali generati dalla unità di controllo. I bit 6:10 e 11:15, che contengono rispettivamente gli indirizzi dei registri rs e rt risultano collegati agli ingressi di indirizzamento del banco dei registri. Conseguentemente il banco presenta in uscita il contenuto dei registri \$rs e \$rt che sono a loro volta presentati in ingresso alla ALU. La ALU esegue la operazione di somma codificato nel campo funct e il risultato è presentato sulla porta di scrittura `data_wr`. Per effetto del segnale di `write_enable`, il valore viene scritto sul registro indirizzato alla porta `addr_wr`, ovvero sul registro specificato nei bit 6:20 che codificano il campo rd.

Paralellamente, l'uscita del program counter è anche collegata all'ingresso del sommatore che incrementa il valore del program counter di +4. Quando il clock segna il prossimo colpo, il valore incrementato viene riscritto sul program counter avviando così l'esecuzione della istruzione successiva.

addI: Se l'istruzione indirizzata dal program counter è `addI $rd,$rs,value`, l'istruzione presentata all'uscita della memoria delle istruzioni ha il formato:

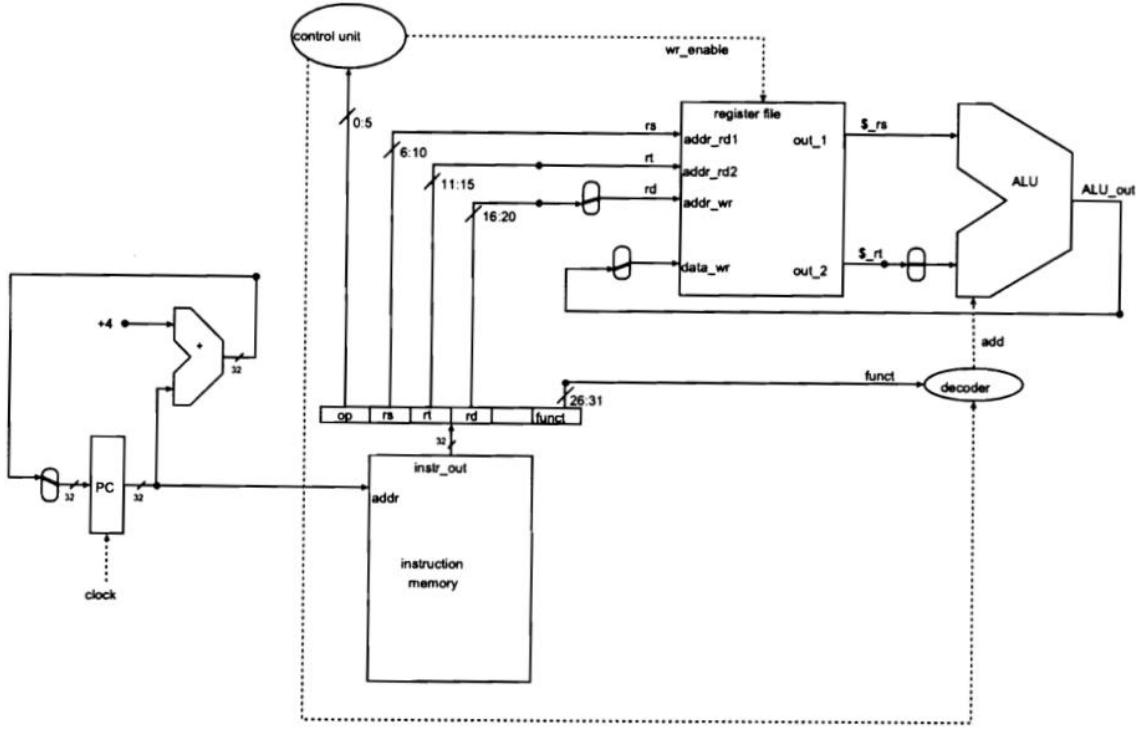


Figura 1.5: Configurazione della CPU nell'esecuzione dell'istruzione **add**.

op=8	rs	rd	value
0:5	6:10	11:15	16:31

Anche in questo caso, i bits del campo operativo sono collegati alla control unit che configura la CPU riducendola alla forma di Fig.1.6.

In questo caso, la ALU prende in ingresso il registro **rs** e il campo immediato **value** esteso a 32 bits. Il risultato della somma è presentato all'ingresso del banco dei registri e scritto sul registro indirizzato alla porta **addr.wr** essendo asserito il segnale **write.enable**.

Si osservi che l'indirizzo del registro di destinazione è codificato sui bits 11:15, diversamente da quanto avviene nel caso dell'istruzione **add** per cui è codificato sui bits 16:20. Questa irregolarità nella codifica del campo destinazione nei formati R ed I rende necessario il multiplexor in ingresso alla porta **addr.wr** del banco dei registri.

In maniera analoga, si osservi che il multiplexor che seleziona il secondo operando della ALU, gestisce i diversi casi nei quali il secondo operando è codificato su un registro (nel formato R) o sul campo immediato (nel formato I).

sw: L'istruzione **sw \$rs, base[\$rt]** è codificata nella forma:

op=8	rt	rs	base
0:5	6:10	11:15	16:31

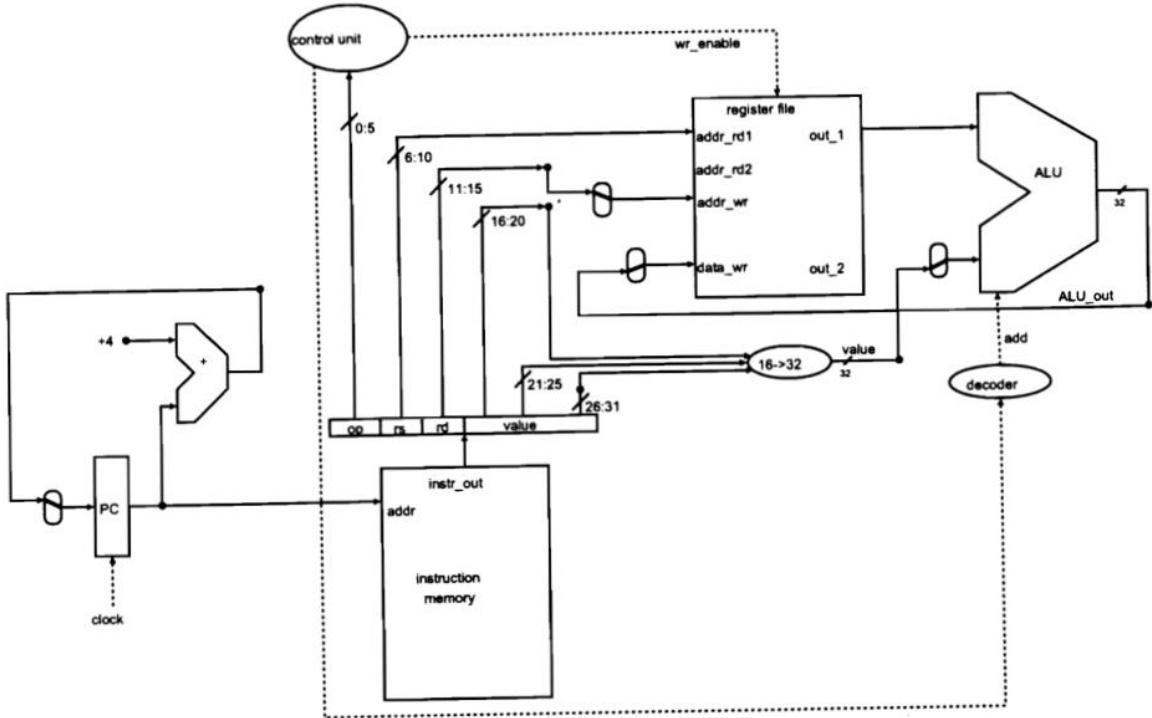


Figura 1.6: Configurazione della CPU nell'esecuzione dell'istruzione `addI`.

In base alla configurazione generata dalla control unit (Fig.1.7), la ALU riceve in ingresso il contenuto del registro indirizzato da `$rt` e il campo immediato `value` esteso a 32 bit. Il risultato della somma dei due operandi, è presentato all'ingresso di indirizzamento della memoria dati. Il segnale di `write_enable` abilita la memoria a scrivere all'indirizzo `base+$rt` il valore presentato sulla porta di ingresso dei dati `data_in`. Tale valore è il contenuto del registro `$rs`.

lw: Nel caso della istruzione `lw $rd, base[$rt]`, la rete assume la configurazione di Fig.1.8 e l'istruzione ha la codifica:

<code>op=8</code>	<code>rt</code>	<code>rd</code>	<code>base</code>
<code>0:5</code>	<code>6:10</code>	<code>11:15</code>	<code>16:31</code>

Come per l'istruzione store word, la ALU riceve in ingresso il contenuto del registro indirizzato da `$rt` e il campo immediato `value` esteso a 32 bit, e la somma dei due valori è presentata all'ingresso di indirizzamento della memoria dati. In questo caso però la memoria dati riceve il segnale `read_enable` per cui ignora il valore presentato sulla porta di scrittura e invece presenta in uscita (sulla porta di lettura `data_out`) il valore contenuto nella locazione indirizzata.

Il valore presentato in uscita dalla memoria dati viene presentato in ingresso

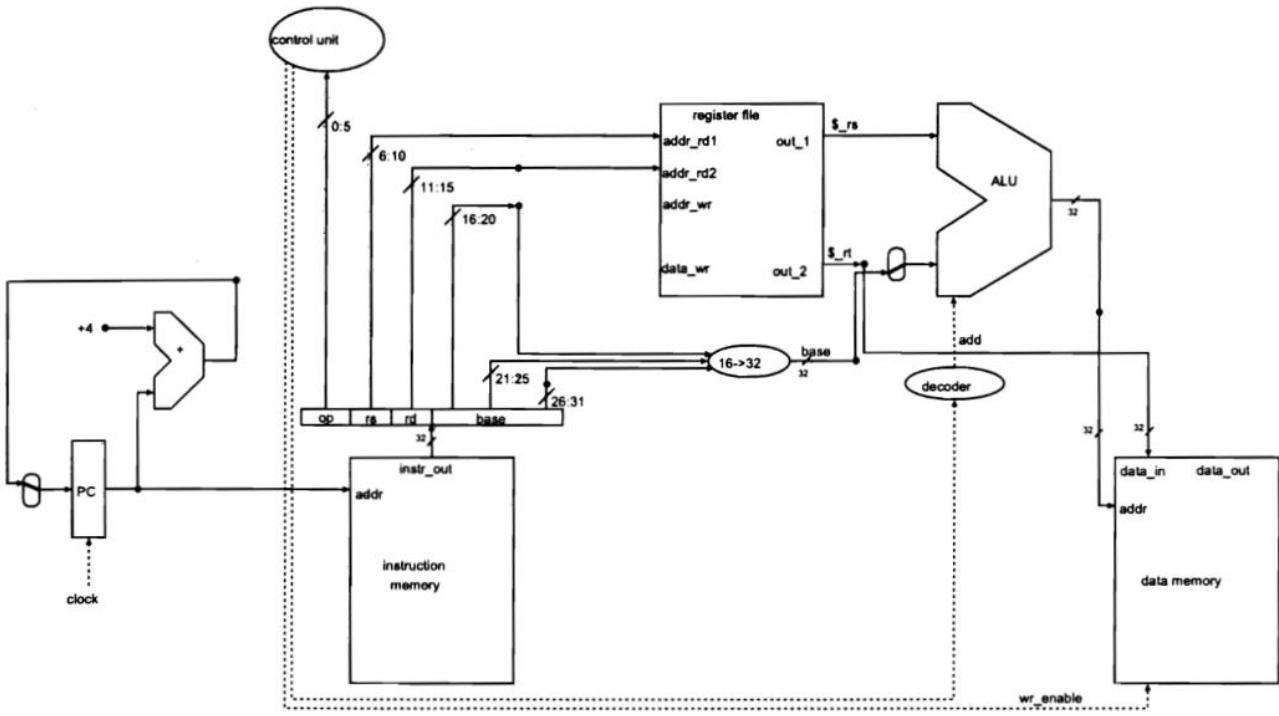


Figura 1.7: Configurazione della CPU nell'esecuzione dell'istruzione **sw**.

alla porta di scrittura del banco dei registri che riceve il segnale di *write_enable* e scrive il valore sul registro indirizzato da **\$rd**.

Si osservi che in questo caso il dato scritto sul banco dei registri proviene dalla memoria dati e non dalla uscita della ALU come avveniva nel caso dell'istruzione aritmetica add. Il multiplexor in ingresso alla porta di scrittura del banco, opportunamente pilotato dalla control unit, assorbe la differenza.

beq: L'istruzione **beq \$rs,\$rt,label** è codificata nella forma:

op=8	rs	rt	label
0:5	6:10	11:15	16:31

La sua esecuzione si distingue da tutte quelle viste fino qui per il modo in cui viene configurato il multiplexor che seleziona la linea che viene portata in ingresso al program counter (vedi Fig.1.9). Il che del resto non stupisce: mentre tutte quelle istruzioni dopo l'esecuzione devono trasferire il controllo alla istruzione successiva, branch-equal ha la *potenzialità* di effettuare un salto in maniera condizionata all'esito di un test di uguaglianza.

La differenza è dovuta al segnale *branch_equal* generato dalla control unit. Per tutte le istruzioni tranne **beq** il segnale è disasserrato e quindi l'uscita della porta AND è sempre disasserrata, di modo che il multiplexor in ingresso al program

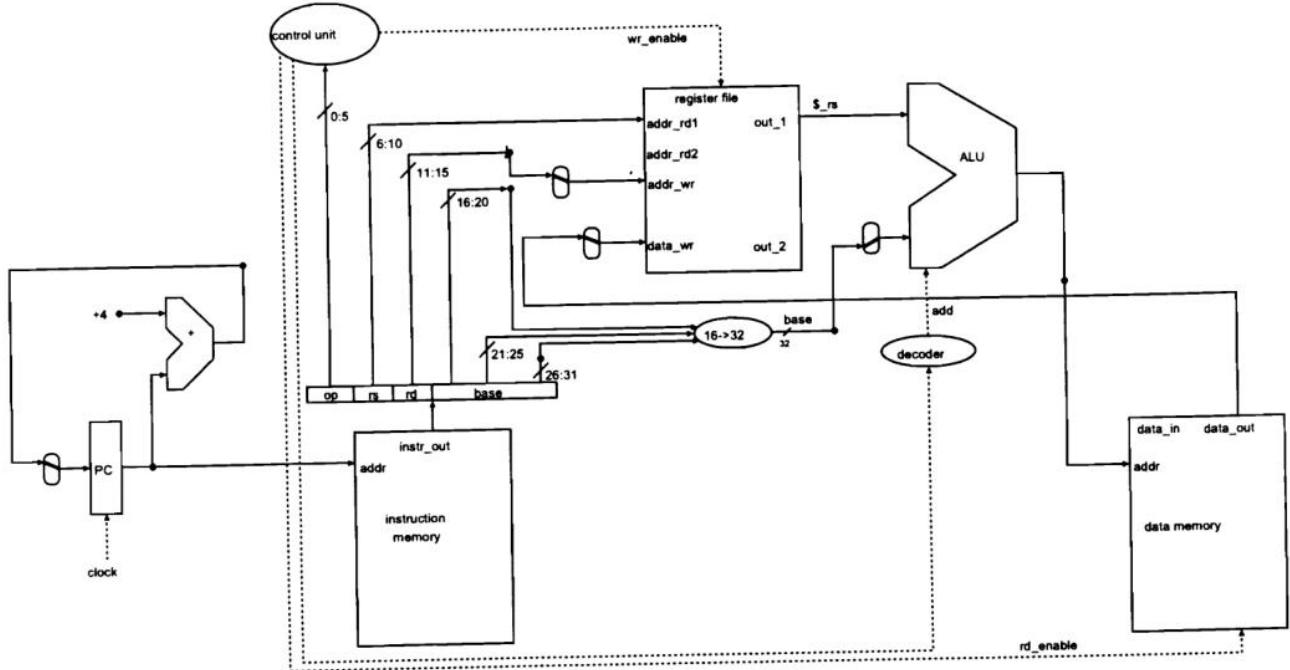


Figura 1.8: Configurazione della CPU nell'esecuzione dell'istruzione `lw`.

counter seleziona la linea che porta il valore $PC+4$. Nel caso dell'istruzione `beq`, il segnale `branch_equal` è invece asserito e quindi lascia che l'uscita dalla porta AND e la selezione della linea in ingresso al program counter siano determinate dal segnale `zero_bit`: se lo `zero_bit` è disasserrato, il segnale portato in ingresso al program counter è $PC+4$; se invece esso è asserito, la linea portata in ingresso al program counter è il risultato dell'operazione $PC+4+label$, il che operativamente corrisponde a eseguire un salto di $label+4$ bytes nella sequenza delle istruzioni.

Lo `zero_bit` viene asserito dalla ALU se l'operazione che essa ha eseguito produce un risultato uguale a zero. In base ai segnali generati dalla control unit, l'operazione eseguita è una differenza e quindi lo `zero_bit` viene asserito se e solo se i contenuti dei due registri `$rs` e `$rt` portati in ingresso alla ALU sono tra loro uguali.

1.3.4 Compilazione, assemblaggio e collegamento

Il processo di rappresentazione di un programma include un numero di trasformazioni dal codice sorgente espresso in un linguaggio di alto livello alla creazione di un programma eseguibile che possa essere caricato in memoria e eseguito sul processore.

Ragionevolmente il programma è inizialmente rappresentato con un linguaggio come può essere il C. Questo è un linguaggio *simbolico*, perché denota operatori e variabili con simboli, ed è di *alto livello* per la capacità di codificare in una unica

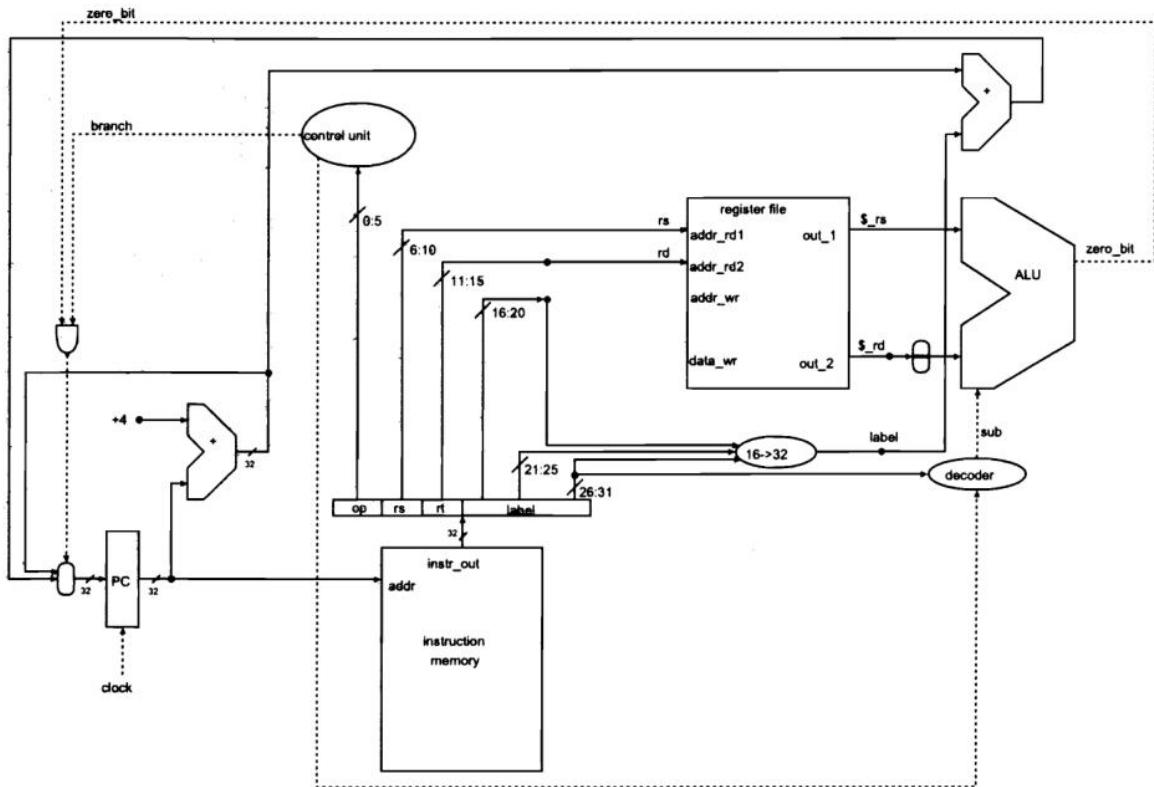


Figura 1.9: Configurazione della CPU nell'esecuzione dell'istruzione `beq`.

istruzione operazioni che richiedono in effetti l'esecuzione di una molteplicità di istruzioni trattate direttamente dal processore.

Il *compilatore* traduce il codice di alto livello in assembler. Questo è ancora un linguaggio simbolico ma di *basso livello*, perché ciascuna delle sue istruzioni può essere eseguita direttamente sul processore. Mentre il linguaggio di alto livello è disegnato per fornire strumenti che permettano una efficace e disciplinata organizzazione del codice, l'assembler è vincolato alle capacità di esecuzione del processore. La distanza tra i due livelli di rappresentazione, il cosiddetto *semantic gap*, è colmato dal compilatore traducendo ciascuna istruzione di alto livello in una molteplicità di istruzioni assembler. Il processo di compilazione non è univoco. Prevede invece scelte che danno spazio ad un complesso problema di ottimizzazione. Questo rende il compilatore un oggetto raffinato e determinante ai fini della efficienza di un programma, tanto che la disponibilità di una varietà di compilatori con buone capacità di ottimizzazione è uno dei fattori determinanti per il successo competitivo di un processore.

L'*assemblatore* traduce il codice assembler in *linguaggio macchina*. La traduzione è 1 a 1, e produce quindi un codice che è ancora di basso livello, ma che è ora *numerico*, essendo le istruzioni codificate come sequenze di numeri. Diversamente dal compilatore, l'assemblatore ha un ruolo banale, che ragionevolmente potrebbe

essere svolto anche da un umano. Il codice numerico generato dall'assemblatore costituisce il programma eseguibile che può essere caricato in memoria ed eseguito dal processore.

Abbiamo fino a qui omesso di rappresentare nel quadro il ruolo del *linker*, la cui definizione rimane astratta fino al momento in cui non sia introdotto il concetto di *funzione* o sottoprogramma. In effetti, nella creazione di un programma può essere conveniente compilare separatamente diverse sezioni del codice collegate tra di loro attraverso istruzioni di trasferimento del controllo, in ultimo realizzate con un *jump*. Questo può avvenire perché le diverse sezioni sono realizzate da parti diverse, o perché esse devono essere modificate separatamente in momenti diversi, o perché una stessa sezione di codice può essere impiegata in più programmi. Nella compilazione separata di ciascuna sezione occorre allora lasciare indeterminati gli indirizzi delle label a cui sono destinate alcune istruzioni di salto. Questi indirizzi di collegamento, gli *hooks*, sono determinati solo al momento in cui le diverse sezioni di codice sono collegate per formare un programma eseguibile. Il linker è il programma che decide come posizionare mutuamente le diverse sezioni di codice, derivando da questo l'informazione necessaria per determinare il valore degli hooks.

Ciascun processore ha un suo linguaggio macchina e quindi un suo assembler. Per questo, un programma compilato per un processore non può in generale essere eseguito su un processore diverso. Fa eccezione il caso in cui l'assembler di un processore include tutte le istruzioni dell'assembler di un processore precedente. Questa condizione di compatibilità all'indietro è realizzata ad esempio nella famiglia dei processori Intel.

Per uno stesso assembler possono esistere una molteplicità di compilatori capaci di tradurre programmi codificati in una varietà di linguaggi di alto livello. Viceversa, un programma scritto in un linguaggio di alto livello può essere compilato verso una varietà di processori target diversi, purché per ciascuno sia disponibile il compilatore.

1.4 Definizione di un linguaggio

La definizione di un linguaggio consiste di sintassi e semantica: mentre la sintassi determina e classifica cosa può apparire in una espressione legale del linguaggio, la semantica definisce il significato associato a ciascuna classe di espressioni legali del linguaggio stesso.

1.4.1 Sintassi di un Linguaggio

Un vocabolario è un insieme di simboli V . Il suo universo linguistico, denotato da V^* , è l'insieme di tutte le sequenze finite di simboli in V . Se ad esempio V contiene i simboli p e c , allora V^* contiene tutte le parole composte con le due lettere, di qualsiasi lunghezza, inclusa la parola nulla, usualmente denotata come λ :

$$V = \{p, c\}$$

$$V^* = \{\lambda, p, c, pp, pc, cp, cc, ppp, ppc, \dots\}$$

Un linguaggio L sul vocabolario V è un sottoinsieme di V^* . Ad esempio, un linguaggio sul vocabolario $V = \{p, c\}$ potrebbe essere costituito da tutte le parole in cui la p può comparire solo in posizioni pari:

$$L = \{\lambda, c, cp, cc, ccc, cccp, \dots\} \subseteq V^*$$

Il problema di definire la sintassi di un linguaggio è quello di definire i limiti del sottoinsieme di V^* . La cosa non è semplice perché a parte i casi banali il linguaggio è composto da infinite espressioni diverse. E queste infinite espressioni diverse devono essere definite attraverso un insieme di regole che ammetta una rappresentazione finita. Esistono vari approcci. Il più comune è quello di usare una grammatica.

1.4.2 Grammatica

Una grammatica è una quadrupla $G = < V, N, S, P >$:

- V è un insieme, detto vocabolario, i cui elementi sono detti simboli terminali.
- N è un insieme, disgiunto da V , i cui elementi sono detti categorie sintatiche.
- S è un elemento di N , detto simbolo iniziale.

- P è una relazione di N su $(N \cup V)^*$, ovvero un sottoinsieme del prodotto cartesiano $N \times (N \cup V)^*$, ovvero un insieme di coppie $\langle n, \eta \rangle$, dove n è un elemento di N , e η è una sequenza di elementi che appartengono a N o a V . Una tale coppia è detta produzione ed è comunemente denotata come $n \rightarrow \eta$.

A questo punto non si dovrebbe ancora avere capito molto di come una grammatica serva a definire un linguaggio. Questo è normale e anche utile a una riflessione: fino a qui sono stati introdotti i simboli che appaiono nella definizione di una grammatica, ma non è ancora stato detto niente sul loro significato: in sostanza è stata definita la sintassi di una grammatica, ma non la sua semantica, ovvero le regole attraverso le quali la quadrupla che rappresenta la grammatica definisce un linguaggio. È un fatto ricorrente nella trattazione dei linguaggi formali: nella descrizione della sintassi, il compito di chi apprende è quello di memorizzare i termini e le categorie che entrano in gioco. Il significato può essere compreso solo nel momento in cui vengono introdotte le regole semantiche associate a ciascuna categoria sintattica.

La regola attraverso la quale una grammatica definisce un linguaggio è descritta nelle seguenti tre definizioni.

- Derivazione diretta: siano pre e $post$ elementi di $(N \cup V)^*$. Sia a_0 un elemento di N , e sia infine α_1 un elemento in $(N \cup V)^*$. Si dice che $pre \alpha_1 post$ deriva-direttamente da $pre a_0 post$, e si scrive $pre a_0 post \xrightarrow{1} pre \alpha_1 post$ se P contiene la produzione $a_0 \rightarrow \alpha_1$.
- Derivazione indiretta: siano α_0 e α_N elementi in $(N \cup V)^*$. Si dice che α_N deriva da α_0 , e si scrive $\alpha_0 \rightarrow \alpha_N$ se esiste una sequenza di elementi $\alpha_i \in (N \cup V)^*$ tali che per $\forall i \in [1, N]$ $\alpha_{i-1} \xrightarrow{1} \alpha_i$.
- Linguaggio definito da una grammatica: il linguaggio L_G definito dalla grammatica $G = \langle V, N, S, P \rangle$ sul vocabolario V è l'insieme degli elementi di V^* che derivano dal simbolo iniziale S attraverso le categorie sintattiche N e le produzioni P .

Esempio

Consideriamo il caso del linguaggio delle espressioni aritmetiche che combinano le variabili di un insieme var attraverso gli operatori $+$, $-$, $*$, \backslash e le parentesi $()$. Giusto per avere un'impressione di quello che vogliamo definire, il linguaggio deve includere tra l'altro l'espressione $a + b * (c - d)$ mentre deve escludere tra l'altro $a + b - *c$.

Conviene iniziare con il vocabolario, nel quale raccogliamo i simboli elementari che possono comparire in una qualsiasi espressione. Questi sono gli elementi dell'insieme var , gli operatori aritmetici e le parentesi tonde:

$$V = var \cup \{+, -, *, /, (,)\}$$

Il passo successivo è la determinazione del simbolo iniziale, ovvero la categoria di cui il linguaggio definisce le espressioni legali. Nel nostro caso è l'espressione aritmetica, che denotiamo con $expr$:

$$S = expr$$

La parte più complessa del processo, è adesso la definizione delle produzioni (P) che permettono di ridurre il simbolo iniziale (S) in simboli terminali passando attraverso un insieme di categorie sintattiche (N). Questa definizione è guidata dalla identificazione delle produzioni mentre le categorie sintattiche sono determinate di conseguenza.

Introduciamo innanzitutto una produzione $expr \rightarrow var$ che permette di derivare un elemento di var dal simbolo iniziale. Il significato di tale produzione è che qualunque variabile in var costituisce di per sé un'espressione legale, ovvero che var è una forma legale per $expr$. Introduciamo poi una produzione $expr \rightarrow (expr)$ che permette di dire che un'espressione legale tra parentesi è essa stessa una espressione legale. Infine introduciamo la produzione $expr \rightarrow expr \ op \ expr$ che permette di dire che la combinazione di due espressioni legali attraverso un operatore è essa stessa un'espressione legale.

In quest'ultimo passo, abbiamo introdotto la categoria sintattica ulteriore op , che nella nostra intenzione rappresenta i simboli $+$, $-$, $*$, e $/$. La definizione di questa categoria richiede 4 ulteriori produzioni $op \rightarrow +$, $op \rightarrow -$, $op \rightarrow *$, e $op \rightarrow /$, che permettono di derivare $+$, $-$, $*$, e $/$ dalla categoria op .

In sintesi, le produzioni della grammatica formano l'insieme P :

$$P = \left\{ \begin{array}{l} expr \rightarrow var, \ expr \rightarrow expr \ op \ expr, \ expr \rightarrow (expr), \\ op \rightarrow +, \ op \rightarrow -, \ op \rightarrow *, \ op \rightarrow / \\ var \rightarrow \dots \end{array} \right\} \quad (1.9)$$

dove per semplicità abbiamo omesso di definire la sintassi con cui sono formati gli identificativi legali per un elemento di var .

Come risultato intermedio della definizione delle produzioni, sono determinate anche le categorie sintattiche, in questo caso la categoria delle espressioni, quella degli operatori e dei riferimenti a variabile:

$$N = \{expr; op; var\}$$

Possiamo a questo punto porci il problema di verificare se nella grammatica che abbiamo scritto la forma $a + b * (c - d)$ è un'espressione legale. Nel nostro caso è un modo di convalidare se la grammatica rappresenta effettivamente il linguaggio che avevamo in testa. Più spesso questo serve a verificare se una espressione è legale rispetto ad una grammatica consolidata, ad esempio quella del linguaggio c.

Assumendo che a, b, c , e d siano simboli in var , possiamo riscrivere l'espressione $a + b * (c - d)$ nella maniera seguente:

$$var + var * (var - var)$$

Applicando ripetutamente la produzione $expr \rightarrow var$ possiamo scrivere

$$var + var * (var - var) \leftarrow expr + expr * (expr - expr)$$

Applicando ora le produzioni $op \rightarrow +$, $op \rightarrow -$ e $op \rightarrow *$ possiamo ora scrivere:

$$expr + expr * (expr - expr) \leftarrow expr \ op \ expr \ op(expr \ op \ expr)$$

Applicando due volte $expr \rightarrow expr \ op \ expr$ e poi $expr \rightarrow (expr)$, questa si riduce a :

$$expr \ op \ expr \ op(expr \ op \ expr) \leftarrow expr \ op \ expr$$

Infine, applicando una ultima volta $expr \rightarrow expr \ op \ expr$ e concatendando le riduzioni, otteniamo:

$$a + b * (c - d) \leftarrow expr$$

che indica che $a + b * (c - d)$ è una espressione legale.

Insistendo ancora sull'esempio, è utile vedere cosa succede nel tentativo di ridurre l'espressione non legale $a + b - *c$. In questo caso, in qualunque ordine siano applicate le produzioni, la riduzione arriva alla forma $a + b - *c \leftarrow expr \ op \ op \ expr$ che non può essere estesa oltre attraverso l'applicazione di alcuna delle produzioni dell'insieme P . In questo caso la riduzione non si è chiusa, e non è quindi vero che $a + b - *c$ deriva dal simbolo iniziale $expr$.

1.4.3 Albero sintattico

Il processo con il quale una sequenza di simboli nel vocabolario V viene ridotta al simbolo iniziale S della grammatica di un linguaggio è convenientemente rappresentato attraverso un *albero sintattico*.

In maniera abbastanza informale, che preciseremo nel capitolo 2.2, un albero è un insieme di nodi sui quali è definita una relazione di successione nella quale

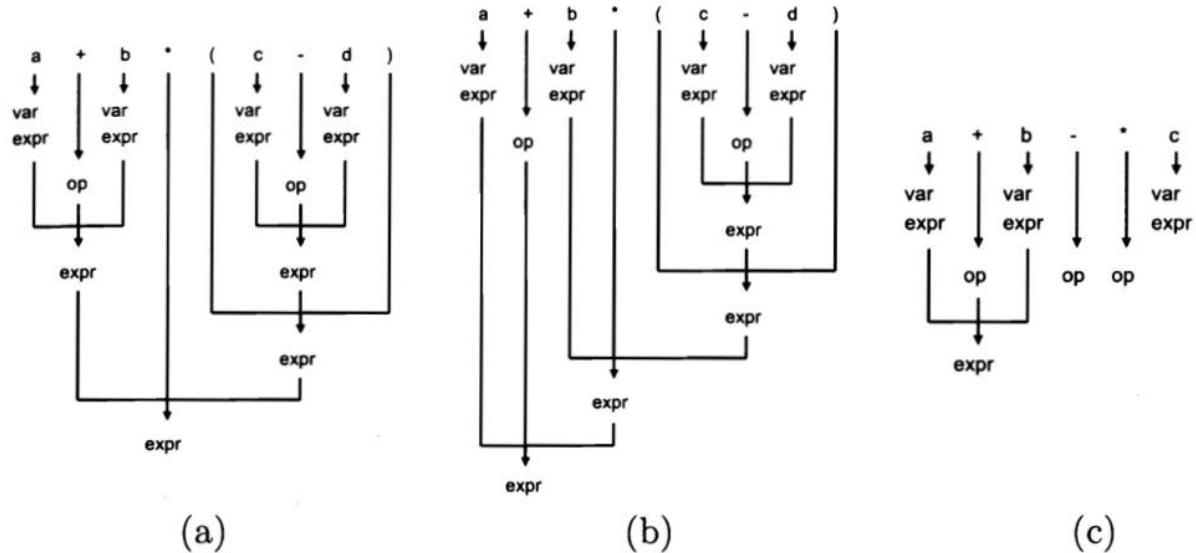


Figura 1.10: Rappresentazione del processo di riduzione di una espressione attraverso l'albero sintattico. **(a-b)** due riduzioni alternative che mostrano che l'espressione $a + b * (c - d)$ deriva dal simbolo iniziale $expr$. **(c)** L'espressione $a + b - *c$ non è una forma legale della categoria sintattica $expr$ non essendo possibile costruire un albero che abbia $expr$ come radice e $a + b - *c$ come sequenza delle foglie.

ciascun nodo ha un unico predecessore e un numero non limitato di successori. Un nodo speciale, detto radice, non ha alcun predecessore.

Nel caso dell'albero sintattico, i nodi rappresentano simboli elementari e produzioni e gli archi rappresentano relazioni di derivazione diretta tra i nodi in base alle produzioni della grammatica. Assegnata una sequenza di simboli nel vocabolario, se sulla sequenza può essere costruito un albero che si chiude su una radice, allora la sequenza costituisce una istanza della categoria sintattica associata alla radice stessa.

Fig.1.10a riporta l'albero sintattico che rappresenta la riduzione $exp \rightarrow a + b * (c - d)$ già discussa nel capitolo 1.4.2. Si osservi che, come illustrato in Fig.1.10b, l'ordine di applicazione delle produzioni che riducono l'espressione non è unico pur conducendo allo stesso risultato finale. In generale, una buona grammatica evita condizioni di ambiguità, tipicamente attraverso regole di precedenza sull'applicazione delle produzioni.

Fig.1.10c mostra un albero sintattico costruito sull'espressione $a + b - *c$. Poiché per tale espressione non è possibile costruire un albero che si chiuda sul simbolo iniziale, l'espressione è illegale, ovvero la sequenza $a + b - *c$ appartiene all'universo linguistico del vocabolario ma non può essere classificata come istanza della categoria $<expr>$.

1.4.4 Il metalinguaggio BNF

L'esempio precedente illustra come la definizione di una grammatica consista essenzialmente nelle sue produzioni. Che sostanzialmente implicano l'insieme delle categorie sintattiche, la selezione del simbolo terminale, e, in ultimo, anche il vocabolario.

Il BNF (Backus Naur Form) è un formalismo che semplifica la rappresentazione e la comprensione delle produzioni rispetto alla notazione elementare $n \rightarrow \eta$. Riflettendoci, il BNF è un linguaggio con cui si rappresentano altri linguaggi, e per questo si dice che è un metalinguaggio.

Del BNF, esistono innumerevoli estensioni e rappresentazioni, in larga parte equivalenti e adattate in aspetti più o meno rilevanti alle caratteristiche di specifici contesti di applicazione. Queste notazioni vanno generalmente sotto il nome di EBNF (extended BNF). Ne vediamo le convenzioni formali più comunemente usate:

- $\eta ::= n$ significa che P contiene la produzione $\eta \rightarrow n$.
- $\eta ::= n_1 | n_2$ significa che P contiene le produzioni $\eta \rightarrow n_1$ e $\eta \rightarrow n_2$. In sostanza permette di esprimere in maniera compatta riduzioni alternative di η .
- $\eta ::= n_{pre}[n]n_{post}$ significa che P contiene le produzioni $\eta \rightarrow n_{pre} n_{post}$ e $\eta \rightarrow n_{pre} n n_{post}$. In sostanza permette di descrivere l'opzionalità del termine n .
- $\eta ::= n_{pre}\{n\}_{min}^{max}n_{post}$ significa che P contiene le produzioni che riducono η in una sequenza aperta dal prefisso n_{pre} , chiusa dal suffisso n_{post} , e la quale contiene un numero di ripetizioni del termine n che può variare tra un minimo min e una massima max . Quando min e max sono omessi, il numero di ripetizioni può assumere qualunque valore non negativo (dunque includendo lo 0).
- Infine, nell'EBNF, simboli terminali e categorie sintattiche sono comunemente distinte attraverso l'uso di parentesi o apici. Noi assumeremo la convenzione che le categorie sintattiche siano raccolte entro i delimitatori $< >$.

Applicando la notazione, le produzioni di Eq.1.9 sono espresse in forma compatta come:

$$\begin{aligned} <expr> ::= & <var> \mid <expr><op><expr> \mid (<expr>) \\ <op> ::= & + \mid - \mid * \mid / \end{aligned} \quad (1.10)$$

Esempio

Per sviluppare un esempio più esteso consideriamo il caso del comando *copy* del sistema operativo DOS. Il comando serve a copiare un file (o almeno serviva per questo fino a tutti gli anni 90 e ai primi anni 00). Per questo il comando specifica il nome, l'unità del filesystem e la posizione del file sorgente e del file destinazione. Un nome è una sequenza di caratteri in formato 8.3: un massimo di 8 caratteri, seguiti opzionalmente da un punto e da una sequenza di al massimo 3 caratteri. L'unità è denotata con una lettera seguita dal carattere due-punti (:). La posizione nel file system, detta anche path, è una sequenza di nomi in formato 8.3 separati dal simbolo di backslash (\), che può iniziare con un nome oppure con un backslash.

La caratteristica che rende interessante l'esempio è che l'interprete dei comandi è capace di gestire l'omissione di alcuni campi del comando assegandogli valori di default. Ad esempio omettendo il nome della destinazione, il file destinazione riceve il nome del sorgente. Se invece è omesso il disco o la posizione, allora questi assumono il valore corrente. Al di là della semantica del comando, ci interessa vedere in che modo il BNF serve alla definizione della sua grammatica.

Chiamiamo *copy – command* il simbolo iniziale, ovvero la categoria sintattica che è definita dalla grammatica. La produzione che la definisce è:

$$< \text{copy} - \text{command} > ::= \text{copy} < \text{file} > [< \text{file} >]$$

copy è un simbolo terminale non essendo racchiuso entro parentesi <>. Viceversa < *file* > è una categoria sintattica e richiede di essere definita attraverso una ulteriore equazione BNF:

$$< \text{file} > ::= [< \text{disk} >:] [< \text{path} >] [< \text{filename} >]$$

< *disk* > è una lettera da *a* a *f*, < *path* > è una sequenza di nomi, preceduto da un backslash opzionale e terminato da un backslash, e < *filename* > è una sequenza di caratteri in formato 8.3:

$$< \text{disk} > ::= a|b|c|d|e|f$$
$$< \text{path} > ::= [\backslash] \{ < \text{filename} > \}$$
$$< \text{filename} > ::= \{ < \text{alphanumeric} > \}_1^8 [.] [\{ < \text{alphanumeric} > \}_1^3]$$

dove < *alphanumeric* > denota un carattere alfabetico, oppure un carattere numerico, oppure un segno meno (-) o un segno di underscore (_).

Una volta identificate le produzioni e il simbolo iniziale, la definizione della grammatica può essere perfezionata raccogliendo le categorie sintattiche e i simboli terminali del vocabolario.

$$V = \text{alphanumeric} \cup \{\text{copy}, :, .\}$$

$$N = \{\langle \text{copy} - \text{command} \rangle, \langle \text{file} \rangle, \langle \text{filename} \rangle, \langle \text{disk} \rangle\}$$

1.4.5 Verifica lessicale, sintattica e contestuale

L'esempio del comando *copy* permette di illustrare un concetto di qualche rilievo. Spesso accade che nella definizione della sintassi di un linguaggio esistano regole che non possono essere convenientemente catturate in una grammatica.

Ad esempio, nella sintassi del comando *copy* è previsto che il nome del file sorgente non possa mai essere omessa, dimodo che l'espressione *copy* non risulta una formulazione legale. Viceversa, in base alla sintassi che abbiamo riportato *copy* senza alcun nome di file è una formulazione legale che risulta dal caso in cui il secondo *<file>* viene omesso e il primo risulta vuoto per l'omissione dei campi *<drive>:*, *<path>* e *<filename>*. Questo limite specifico potrebbe essere superato con una descrizione più complessa che distinguesse la sintassi del *<file>* destinazione da quello del *<file>* sorgente, in maniera da imporre che nel *<file>* destinazione il *<filename>* non possa mai essere omesso:

```
<copy-command> ::= copy <source-file> [<dest-file>]
<source-file> ::= [<disk>:] [<path>] <filename>
<dest-file> ::= [<disk>:] [<path>] <filename>
```

Esistono casi in cui rappresentare un vincolo del linguaggio nella grammatica richiede un aumento di complessità molto maggiore, quando non è addirittura impossibile entro la capacità espressiva della grammatica. Ad esempio, nella sintassi del comando *copy* è specificato che il *<dest-file>* deve essere diverso dal *<source-file>*; come anche è richiesto che se il *<dest-file>* è omesso allora il *<source-file>* deve includere un *<path>*.

Per vincoli di questo tipo è conveniente ammettere che la definizione della sintassi sia completata da cosiddetti vincoli contestuali. Questi sono espressi più o meno formalmente come annotazioni che complementano la grammatica. L'uso di vincoli contestuali riduce la non-ambiguità di un linguaggio e ne ostacolano il trattamento automatico attraverso programmi compatti. Tuttavia, tali tipi di vincoli permettono di esprimere linguaggi che non sarebbero rappresentabili con una grammatica e comunque permettono di ottenere grammatiche compatte e più facilmente comprensibili.

Con l'introduzione dei vincoli contestuali, il processo di interpretazione di una espressione si articola su 3 fasi successive:

- nella verifica lessicale si verifica che l'espressione appartenga all'universo linguistico V^* del vocabolario, ovvero che tutti i simboli che compaiono nell'espressione appartengono al vocabolario V .

Ad esempio, *copy pippo?pluto* non è una formulazione legale del comando *copy* perché il carattere ? non appartiene al vocabolario del linguaggio con cui può essere formato il comando.

- nella verifica sintattica, una espressione lessicalmente legale viene analizzata per verificare se è una riduzione legale del simbolo iniziale del linguaggio.

Ad esempio *copy pippo pluto paperino* è lessicalmente ma non sintatticamente corretta essendo riducibile alla forma $<\text{copy-command}> <\text{file}>$ ma non al simbolo iniziale $<\text{copy - command}>$.

- infine, nella verifica contestuale, espressioni che sono lessicalmente e sintatticamente corrette sono comparate contro i vincoli contestuali.

Come osservavamo, *copy pippo* è sintatticamente corretto ma non soddisfa il vincolo contestuale per cui se è omesso il $<\text{file}>$ di destinazione allora il $<\text{file}>$ sorgente deve specificare il $<\text{path}>$.

1.4.6 Semantica e sintassi

La grammatica che determina la sintassi di un linguaggio fornisce anche la base su cui è possibile innestare una definizione non ambigua della semantica. In questo caso, la definizione della semantica impiega le produzioni della grammatica per ridurre il significato di un'espressione alla composizione del significato delle parti che la compongono. Il che va sotto il nome di induzione strutturale. Seguiremo questo approccio in maniera estensiva nella descrizione del linguaggio c. Per il momento illustriamo il concetto considerando un esempio.

Consideriamo il caso delle espressioni della logica Booleana. Una tale espressione, che denotiamo con la categoria $<\text{expr}>$ è ottenuta combinando ricorsivamente variabili di tipo Booleano che assumono valori *true* e *false* attraverso i connettivi di congiunzione \wedge (and logico), disgiunzione \vee (or logico), e negazione \neg (not logico), e con l'uso delle parentesi (). Denotando con var l'insieme delle variabili Booleane:

$$\begin{aligned} <\text{expr}> ::= & \text{ var} | (<\text{expr}>) \\ & \neg <\text{expr}_1> | <\text{expr}_1> \wedge <\text{expr}_2> | <\text{expr}_1> \vee <\text{expr}_2> \end{aligned}$$

La semantica di un'espressione $<\text{expr}>$ consiste nel valore Γ che essa restituisce. La definizione della semantica richiede allora che per ciascuna diversa riduzione di

$<expr>$ sia definito il valore di $\Gamma(<expr>)$ in funzione del valore restituito dalle eventuali espressioni che la compongono. Questo è specificato attraverso una serie di clausole che coprono tutte le diverse riduzioni di $<expr>$:

- se $<expr> ::= var$, allora $\Gamma(<expr>)$ è il valore della variabile var . Poiché var è una variabile Booleana tale valore è *true* oppure *false*.
- se $<expr> ::= (<expr_1>)$, allora $\Gamma(<expr>) = \Gamma(<expr_1>)$, ovvero il valore restituito dall'espressione è lo stesso valore restituito dall'espressione entro parentesi.
- se $<expr> ::= \neg <expr_1>$, allora $\Gamma(<expr>) = \neg \Gamma(<expr_1>)$. A sua volta la semantica del connettivo \neg può essere definita per enumerazione dei casi:

X	$\neg X$
<i>true</i>	<i>false</i>
<i>false</i>	<i>true</i>

- se $<expr> ::= <expr_1> \wedge <expr_2>$, allora $\Gamma(<expr>) = \Gamma(<expr_1>) \wedge \Gamma(<expr_2>)$. Anche in questo caso, la semantica del connettivo di congiunzione può essere definita con un tabella delle verità:

X	Y	$X \wedge Y$
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>
<i>false</i>	<i>true</i>	<i>false</i>
<i>false</i>	<i>false</i>	<i>false</i>

- se $<expr> ::= <expr_1> \vee <expr_2>$, allora $\Gamma(<expr>) = \Gamma(<expr_1>) \vee \Gamma(<expr_2>)$, dove la semantica del connettivo di disgiunzione è definita dalla seguente tabella delle verità:

X	Y	$X \vee Y$
<i>true</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>false</i>	<i>false</i>

Possiamo ora considerare come esempio concreto una espressione:

$$a \wedge (b \vee \neg c)$$

Fig.1.11 ne illustra l'albero sintattico, che indica che $a \wedge (b \vee \neg c)$ è un'espressione legale. Si osservi che sull'albero viene anche valutata la semantica dell'espressione

nel caso in cui le variabili a , b , e c siano interpretate sui valori $a = \text{true}$, $b = \text{false}$, $c = \text{false}$. Il calcolo è ottenuto applicando le clausole semantiche in maniera da attribuire un valore Γ a ciascuna delle espressioni intermedie che appaiono nell'albero sintattico, partendo dall'alto con i valori di a , b , e c , e scendendo fino a raggiungere la radice dell'albero.

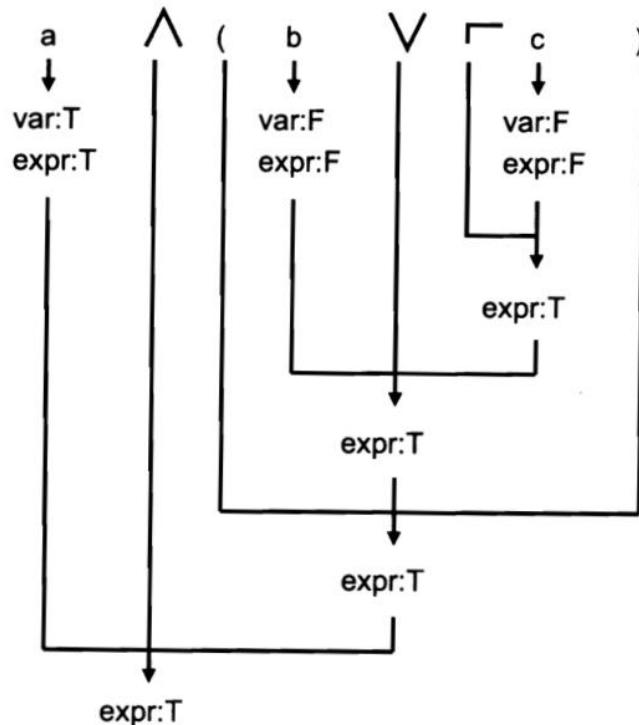


Figura 1.11: L'albero sintattico per l'espressione Booleana $a \wedge (b \vee \neg c)$ annotato con i valori restituiti dalle espressioni intermedie (T denota True e F False) nel caso in cui $a = \text{true}$, $b = \text{false}$, $c = \text{false}$. L'albero mostra che sotto tale interpretazione delle variabili, l'espressione restituisce il valore vero.

1.5 Il linguaggio c

In questo capitolo viene definito il linguaggio c.

La trattazione fa leva congiuntamente sui capitoli precedenti, con diversi gradi di dipendenza: la descrizione informale del frammento di linguaggio c nel capitolo 1.1 aiuta la comprensione fornendo una visione di insieme, ma è completamente coperta da quanto viene ora esposto; la trattazione di come un programma c può essere compilato in assembler, tradotto in linguaggio macchina ed eseguito su un processore fornita nei capitoli 1.2 e 1.3 fornisce una ragione per alcune caratteristiche del linguaggio, ma può essere anche omessa; gli elementi di teoria dei linguaggi forniti nel capitolo 1.4 sono invece usati diffusamente e risultano strettamente necessari per la comprensione.

1.5.1 Tipi, variabili e costanti

Tipi elementari

In un linguaggio di programmazione un tipo è caratterizzato dall'insieme dei valori rappresentati e dalle operazioni che possono essere applicate su di essi.

Ai fini dell'implementazione il tipo è caratterizzato anche dal modo con cui i valori sono codificati e dagli algoritmi che sono usati nella realizzazione delle operazioni. Quest'ultimo aspetto resta per lo più trasparente al programmatore a parte le sue ultime conseguenze sulla dinamica dei valori e sugli effetti di precisione finita nelle operazioni.

Il c include 5 tipi elementari: `char`, `int`, `float`, `double` e `void`. Al tipo `int` possono essere applicati optionalmente i modificatori `short`, `long` e `unsigned`:

```
<type>::=char | [unsigned] [short | long] int | float | double | void
```

- `char` rappresenta i caratteri della tabella ASCII.
- `int` rappresenta i numeri interi con segno. Il numero di bytes usati nella rappresentazione, e quindi la dinamica dei valori, può variare a seconda dell'architettura della macchina su cui viene compilato il programma. Tipicamente sono usati 4 bytes.

Al tipo intero possono essere applicati i modificatori `long`, `short`, e `unsigned`. Il modificatore `long` aumenta il numero di bytes usati nella codifica, espandendo la dinamica dei valori rappresentati; viceversa `short` riduce il numero di bytes della codifica. Ovviamente `long` e `short` sono alternativi. Infine, `unsigned int` non rappresenta il segno e raddoppia la dinamica dei valori positivi.

L'effetto concreto dei modificatori `short` e `long` dipende dal dall'architettura della macchina su cui viene compilato il programma. In generale è garantito solo che il numero di bytes che codificano uno `short int` sia minore o uguale di quello che codifica un `int` e che questo sia minore o uguale del numero di bytes che codificano un `long int`. Tipicamente, su un PC uno `short` è codificato su 1 byte e un `long` su 4.

L'uso dei modificatori, in congiunzione con la dipendenza della rappresentazione rispetto all'architettura della macchina, attribuiscono agli interi del c una complessità apparentemente superiore al necessario. Al di là dei problemi minori legati all'apprendimento del linguaggio, questo richiede accorgimenti nella realizzazione di programmi che devono potere eseguire su macchine con diverse architetture.

- `float` rappresenta valori razionali in formato floating point.
- `double` rappresenta valori razionali in formato floating point in doppia precisione.
- Il tipo `void` è il tipo nullo, e non esistono variabili di tipo `void`. Il tipo `void` serve in realtà a specificare un tipo laddove questo non ha nessun particolare effetto ma permette di mantenere regolarità nel linguaggio. In sostanza è qualcosa che gioca il ruolo di un tipo dal punto di vista sintattico ma non da quello semantico. Il che può risultare sostanzialmente incomprensibile a questo punto, ma diventerà chiaro quando saranno trattati puntatori e funzioni.

Variabili

Una variabile è una locazione di memoria che contiene un valore di un tipo. Il valore può essere modificato nel corso della computazione, mentre il tipo è invariante. Concettualmente, è conveniente pensare che anche la locazione di memoria sia invariante. Nello studio dei sistemi operativi si potrà scoprire che le variabili possono essere rilocate, tale rilocazione è però del tutto trasparente e inaccessibile al programmatore.

Una variabile è associata a un nome. È utile osservare che il fatto di potere referenziare una variabile attraverso un nome è uno degli aspetti caratterizzanti di un linguaggio simbolico, come è il c e come è l'assembler. In una prospettiva di più basso livello, quella del linguaggio macchina, la variabile è invece identificata dall'indirizzo della sua locazione di memoria. Facendo riferimento alla trattazione del capitolo 1.3 questo indirizzo è il valore del campo `Astart` che compare nella rappresentazione numerica delle istruzioni `lw` e `sw`.

Ogni variabile deve essere *dichiarata* prima dell'uso, specificandone il nome e il tipo. La dichiarazione ha il seguente formato:

```
<declaration> ::= <type> <decl>;  
<decl> ::= <identifier> | ...
```

dove *<identifier>* è il nome della variabile. Tale nome è formato come una sequenza di caratteri alfabetici e numerici, con possibile uso del segno di underscore (-) tranne che nel primo carattere. Il nome può avere una lunghezza arbitraria anche se il compilatore non distingue nomi che coincidono sui primi 32 caratteri.

Si osservi che nella produzione della categoria *<decl>* abbiamo lasciato dei punti di sospensione, per anticipare che la produzione sarà estesa quando arriveremo a trattare puntatori, array, funzioni, e strutture.

In generale, la semantica di una dichiarazione *<declaration>* consiste nell'introdurre un simbolo associato a un nome e un tipo nello spazio degli oggetti referenziabili nel programma. Nel caso particolare di una dichiarazione della forma *<type> <identifier>*; il simbolo denota una variabile di tipo *<type>* associata al nome *<identifier>*. Il compilatore anche provvede ad allocare in memoria lo spazio in cui la variabile viene mantenuta.

Una variabile dichiarata può essere *referenziata* attraverso il suo identificatore:

```
<var> ::= <identifier> | ...
```

In generale, la semantica di un riferimento a variabile *<var>* consiste nell'individuare la variabile denotata dal riferimento. In particolare, il riferimento *<identifier>* denota la variabile associata al nome *<identifier>* da qualche precedente dichiarazione.

Ancora, si osservi che i punti di sospensione nella espressione delle produzioni della categoria *<var>* indicano che questa verrà estesa nel seguito, con l'introduzione di puntatori, array, funzioni, e strutture. Per il momento ci limitiamo ad aggiungere che un riferimento a variabile può essere raccolto entro parentesi per condizionare la priorità di associazione:

```
<var> ::= <identifier> | (<var>) | ...
```

Costanti

Le costanti denotano valori che non sono modificati nel corso dell'elaborazione. Anche le costanti hanno un tipo. Il che può sfuggire al programmatore, ma non al compilatore che deve comunque decidere il formato della loro rappresentazione e la specifica istruzione assembler che usa l'unità aritmetica del processore appropriata per il loro trattamento.

Il tipo associato alla costante può essere dichiarato esplicitamente, o è derivato dal compilatore in base al formato con cui la costante è espressa nel programma. Omettiamo una descrizione dettagliata e ci limitiamo ad alcuni esempi che illustrano il più della questione: 12 è una costante intera; `unsigned 12` è una costante intera senza segno; 12. è una costante float; `double 12` è una costante float in doppia precisione; 'a' è un carattere; `0xAF12B` è una costante esadecimale (valore AAF12B).

In linea di principio la costante non è memorizzata in una locazione di memoria, ma va piuttosto intesa come una parte del codice.

Facendo riferimento alla trattazione del linguaggio macchina sviluppata nel capitolo 1.3, le costanti sono codificate nel campo immediato di istruzioni come `addI`. Per ragioni tecnologiche legate all'ampiezza limitata disponibile per gli operandi immediati, il compilatore può mappare la rappresentazione di una costante su una variabile che rimane non visibile al programmatore.

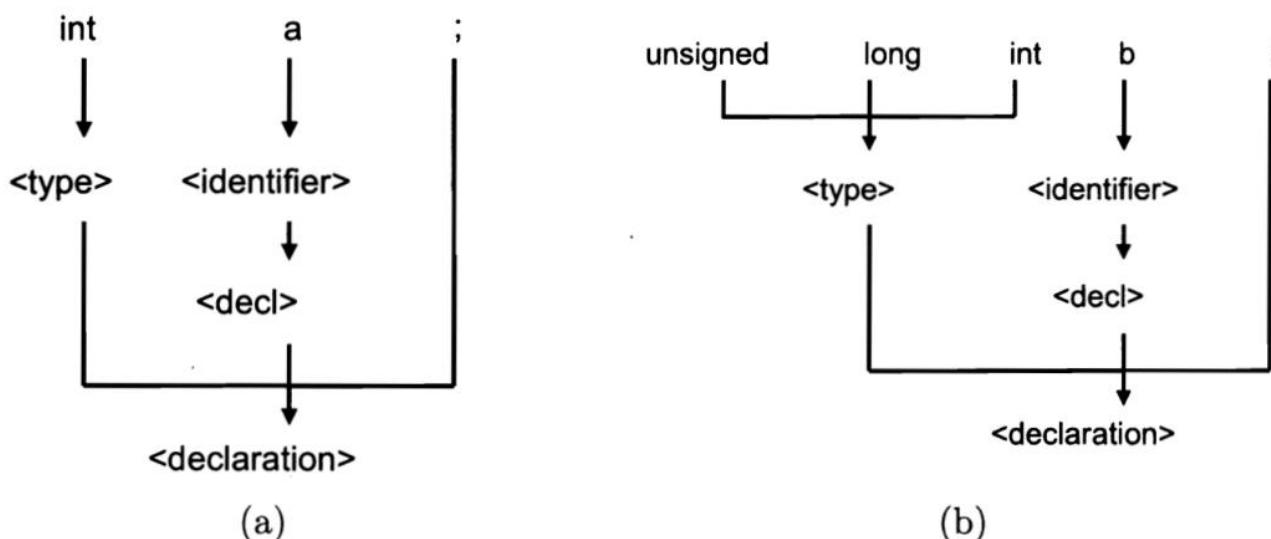


Figura 1.12: L'albero sintattico delle dichiarazioni `int a;` e `unsigned long int b;`.

Esempi

Consideriamo le due dichiarazioni:

```
int a;
unsigned long int b;
```

Per verificarne la legalità occorre verificare che esse possono essere ridotte al simbolo delle dichiarazioni di variabile, che abbiamo denotato con `<declaration>`.

Vediamo in che modo, con l'intento di richiamare notazione e significato delle grammatiche più che di approfondire la comprensione delle dichiarazioni del c, a questo punto abbastanza elementari.

Fig.1.12a riporta l'albero sintattico che dimostra che la dichiarazione int a; può essere ridotta al simbolo <*declaration*>. In particolare si osservi che nella riduzione int ← <*type*> sono omessi entrambi i campi opzionali *unsigned* e *short|long* della produzione

```
<iotype> ::= ... | [unsigned] [short|long] int|...
```

L'albero sintattico di Fig.1.12b riporta la riduzione della dichiarazione *unsigned long int b;*. In questo caso la categoria Type è derivata includendo entrambi i campi opzionali *unsigned* e *short|long*.

1.5.2 Operatori ed espressioni

Le espressioni combinano variabili e costanti attraverso operatori.

Nella presentazione della sintassi sarebbe conveniente raggruppare gli operatori in base al numero e la posizione degli operandi, distinguendo operatori unari-prefissi, unari-suffissi, binari, e ternari. Dal punto di vista semantico è invece conveniente raggruppare gli operatori in categorie che distinguono operatori aritmetici, logici, relazionali, sui bit, di assegnamento, di composizione. Seguiremo questo secondo approccio, il più consueto, che richiede una enumerazione più estesa delle produzioni, ma semplifica la descrizione della semantica.

Sintassi

I termini elementari delle espressioni sono variabili e costanti, che sono di per sé espressioni legali. Sulle variabili possono essere applicati gli operatori di incremento ++ e decremento --, sia in forma prefissa che suffissa:

```
<expr> ::= <const> | <var> | ++<var> | --<var> | <var>++ | <var>-- | ...
```

Ci sono poi un numero di operatori attraverso i quali la combinazione di espressioni produce un'altra espressione. In particolare ce ne sono 5 di tipo aritmetico (somma +, sottrazione -, prodotto *, divisione / e modulo %), 6 di tipo relazionale (minore <, minore o uguale <=, uguale ==, diverso !=, maggiore o uguale >=, e maggiore >), 3 di tipo logico (congiunzione (and logico) &&, disgiunzione (or logico) ||, e negazione !), e 5 di manipolazione dei bit (and bit-a-bit & , or bit-a-bit |, negazione bit-a-bit ~, shift a destra >> e a sinistra <<). L'operatore logico di negazione e l'operatore di negazione bit-a-bit sono operatori unario prefissi, tutti gli altri sono operatori binari:

$\langle expr \rangle ::= \dots | \langle expr_1 \rangle \ op_2 \ \langle expr_2 \rangle | op_1 \ \langle expr_2 \rangle | \dots$

$\langle op_2 \rangle ::= + | - | * | / | \% |$
 $< | \leq | == | != | \geq | > |$
 $\&& | \mid |$
 $\& | \mid | \gg | \ll$

$\langle op_1 \rangle ::= ! | \sim$

In C, l'assegnamento del risultato di un'espressione a una variabile è esso stesso un operatore. Anche se la cosa può apparire non intuitiva, ha il pregio di semplificare largamente la sintassi delle istruzioni. L'assegnamento ammette una forma compatta in cui un operatore binario è prefisso al segno di uguale:

$\langle expr \rangle ::= \dots | \langle var \rangle = \langle expr_1 \rangle | \langle var \rangle \langle op_2 \rangle = \langle expr_2 \rangle | \dots$

Per raccogliere, concatenare, sequenzializzare espressioni esistono poi le parentesi tonde (), la virgola , e l'operatore ternario ? ::

$\langle expr \rangle ::= \dots | (\langle expr_1 \rangle) | \langle expr_1 \rangle, \langle expr_2 \rangle | \langle expr_1 \rangle ? \langle expr_2 \rangle : \langle expr_3 \rangle | \dots$

Infine c'è il cast che permette di controllare il tipo con cui è rappresentato il risultato di un'espressione:

$\langle expr \rangle ::= \dots | (\langle cast-type \rangle) \ \langle expr_2 \rangle | (\langle cast-type \rangle \ \langle expr_2 \rangle) | \dots$
 $\langle cast-type \rangle ::= \langle type \rangle | \dots$

Si noti che per ora la specifica $\langle cast-type \rangle$ del tipo del cast è un tipo elementare nella categoria $\langle type \rangle$. Con l'introduzione dei puntatori dovremo estendere la sintassi.

L'elenco delle forme di un'espressione per ora finisce qui. Lasciamo aperti i punti di sospensione perché ci manca ancora di aggiungere qualcosa che introdurremo con la trattazione dei puntatori e delle funzioni.

Semantica

La semantica di un'espressione consta di due componenti: il valore restituito dall'espressione, e l'effetto che il calcolo dell'espressione produce sul valore delle variabili del programma.

Tutte le espressioni restituiscono un valore. È utile sottolineare ancora che tale valore è espresso in un tipo. Che condiziona l'applicabilità di alcuni operatori e la loro semantica, ovvero, più concretamente, la loro traduzione in istruzioni assembler e la loro esecuzione in unità aritmetiche della CPU.

Solo le operazioni di incremento e decremento delle variabili (`++` e `--`) e l'operazione di assegnamento (`=`) producono effetto sulle variabili. L'effetto sulle variabili è chiamato side effect, che significa effetto collaterale. Il termine suggerisce che i side effects siano una questione minore, fastidiosa ma non evitabile. Viceversa i side-effects sono una componente intenzionale e strutturale della programmazione c, essendo l'unico mezzo con il quale è possibile elaborare i dati memorizzati nelle variabili.

Descriviamo la semantica delle espressioni distinguendo le diverse forme della sintassi secondo un meccanismo di induzione strutturale.

- Il riferimento a una variabile o una costante ($\langle expr \rangle ::= \langle const \rangle | \langle var \rangle$) restituiscono il valore della variabile o della costante (nel loro tipo) senza produrre side-effects.
- Gli operatori di incremento e decremento ($\langle expr \rangle ::= \langle var++ \rangle | \langle ++var \rangle | \langle var-- \rangle | \langle --var \rangle$) modificano la variabile a cui sono applicati: gli operatori di incremento aggiungono il valore 1, quelli di decremento lo sottraggono, indipendentemente dalla applicazione in forma prefissa o suffissa.

La differenza tra le due forme attiene invece al valore restituito: mentre l'operatore prefisso restituisce il valore incrementato, quello suffisso restituisce il valore come era nella variabile prima dell'operazione. Denotando con $\Gamma(\langle expr \rangle)$ il valore restituito da un'espressione, questo è catturato nelle regole:

$$\Gamma(\langle var \rangle++) = \Gamma(\langle var \rangle) + 1$$

$$\Gamma(\langle var \rangle--) = \Gamma(\langle var \rangle) - 1$$

$$\Gamma(\langle var \rangle++) = \Gamma(\langle var \rangle)$$

$$\Gamma(\langle var \rangle--) = \Gamma(\langle var \rangle)$$

Alcuni esempi possono aiutare la comprensione. Se `x` vale 12: l'espressione `x` restituisce il valore 12 e non modifica la variabile; `++x` restituisce il valore 13 e assegna il valore 13 a `x`; l'espressione `x++` restituisce il valore 12 e assegna il valore 13 a `x`.

- La composizione di espressioni con operatori (($\langle expr \rangle ::= \langle expr_1 \rangle \langle op_2 \rangle \langle expr_2 \rangle | \langle op_1 \rangle \langle expr_2 \rangle$)) non aggiunge side-effects a quelli già indotti dalle espressioni composte.

Il valore restituito si ottiene applicando la semantica dell'operatore specifico ai valori restituiti dalle espressioni composte:

$$\Gamma(\langle expr_1 \rangle \langle op_2 \rangle \langle expr_2 \rangle) = \Gamma(\langle expr_1 \rangle) \langle op_2 \rangle \Gamma(\langle expr_2 \rangle)$$

$$\Gamma(<op_1><expr_2>) = <op_1>\Gamma(<expr_2>)$$

Gli operatori aritmetici di somma, sottrazione, prodotto, divisione ($<op_2> ::= + \mid - \mid * \mid /$) fanno quello che uno si aspetta. Vale la pena di ricordare comunque che le operazioni sono eseguite in aritmetica finita, con la possibilità di overflow/underflow e/o di errori di troncamento. E vale anche la pena di ricordare che, contrariamente all'intuizione, operatori che hanno la stessa espressione sintattica possono avere una diversa semantica a seconda del tipo delle variabili a cui sono applicati. Per fare un esempio, la somma sugli interi e quella sui float, che sono entrambe espresse con il segno +, vengono compilate usando istruzioni assembler diverse, che sono eseguite con algoritmi diversi in diverse unità aritmetiche del processore. Per effetto di questo, la somma tra interi fornisce un risultato esatto a meno che non produca un overflow, mentre la somma tra float è sempre affetta da un errore di approssimazione ma è quasi impossibile che incorra in un overflow.

L'operatore aritmetico di modulo ($<op_2> ::= \%$) restituisce il resto della divisione intera tra gli operandi. Ad esempio $14\%3$ restituisce 2, essendo 2 il resto della divisione di 14 per 3.

Per tutti gli operatori aritmetici, il tipo del valore restituito è il tipo restituito dalle espressioni che restituiscono gli operandi.

Nel caso in cui gli operandi abbiano valore espresso in tipi diversi, la semantica dell'operazione include la conversione dell'operando più debole al formato del più forte. Dove la forza corrisponde sostanzialmente all'ampiezza dell'insieme dei valori rappresentati, con una sorta di ordinamento principale dei tipi in cui `char < int < float < double` e con alcuni vincoli ausiliari quali `unsigned int < int < short int < int < long int`. L'operazione è eseguita nel formato dell'operatore più forte e il valore restituito è espresso anch'esso in tale formato.

Ad esempio se `n` è un intero e `x` è un float, nella somma `x+n`, il valore della variabile `n` viene convertito nella rappresentazione float, la somma è eseguita con le regole dei float, e il valore restituito è anch'esso un float.

Gli operatori relazionali ($<op_2> ::= > | >= | == | != | <= | <$) servono a testare condizioni di ordinamento e uguaglianza tra valori. Restituiscono il valore 1 e 0 (nel tipo intero) a seconda che la relazione testata sia verificata o falsificata. Se gli operandi hanno tipo diverso, prima della operazione il tipo più debole è convertito al formato più forte.

Per esemplificare, se `x` vale 12: l'espressione `x==10` restituisce il valore 0; l'espressione `x>10` restituisce il valore 1; l'espressione `x!=10` restituisce il valore 1.

Vale la pena di ricordare che in C non esiste il tipo Booleano, che è invece sostituito dagli interi con la convenzione che qualunque valore diverso da 0 è interpretato come vero e il valore 0 è interpretato come falso. sotto questa convenzione è ragionevole dire che un operatore relazionale restituisce vero o falso a seconda che la relazione che esso denota sia soddisfatta o meno dagli operandi.

Gli operatori logici ($\langle op_2 \rangle ::= \&\& | || | !$) realizzano i connettivi della logica Booleana. $X \&\& Y$ restituisce 1 (vero) se X e Y valgono entrambi un valore diverso da 0, e restituisce 0 (falso) se uno almeno tra X e Y vale 0. Viceversa, $X || Y$ restituisce vero se uno almeno tra X e Y è vero, e restituisce falso se X e Y sono entrambi falso. Infine $!Y$ restituisce vero se Y è falso, e restituisce falso se Y è vero. Gli operandi possono essere espressi in qualsiasi tipo, mentre il risultato è sempre restituito come intero.

È interessante soffermarsi sui side-effects dell'espressione $expr_1 \&\& expr_2$: l'espressione $expr_1$ viene calcolata e quindi applicati i suoi effects; se $expr_1$ restituisce vero allora viene anche calcolata $expr_2$ e applicati i suoi side effects; se invece $expr_1$ ha restituito falso, allora la congiunzione è già determinata, per cui viene omesso il calcolo di $expr_2$ e di conseguenza la applicazione dei suoi side effects. L'operatore di disgiunzione $||$ ha una semantica analoga: nell'espressione $expr_1 || expr_2$, se $expr_1$ restituisce vero, allora $expr_2$ non viene calcolato e i suoi side effects non sono quindi applicati. Questa asimmetria, nota con il nome di *cortocircuito* (short-circuit) viene comunemente usata nella pratica della programmazione quando esistono due condizioni delle quali la seconda può essere testata solo nel caso in cui la prima restituisca vero. Non è una buona pratica laddove le espressioni composte producano side-effects.

Omettiamo di descrivere gli operatori sui bit ($\langle op_2 \rangle ::= \& | \mid | \& | \mid | !$). Non ne facciamo uso in questo testo, ed è abbastanza difficile che diventino utili prima che il programmatore abbia l'esperienza sufficiente per comprenderli autonomamente.

- L'espressione di assegnamento $\langle var \rangle = \langle expr_2 \rangle$ calcola l'espressione $\langle expr_2 \rangle$ e ne assegna il valore alla variabile $\langle var \rangle$, dopo averlo convertito al tipo della variabile $\langle var \rangle$ stessa.

Questo produce prima i side effects dovuti al calcolo dell'espressione e poi modifica la variabile con l'assegnamento. Il valore restituito è lo stesso che viene assegnato alla variabile e in particolare ha il tipo della variabile:

$$\Gamma(\langle var \rangle = \langle expr_2 \rangle) = \Gamma(\langle expr_2 \rangle)$$

Il fatto che un assegnamento sia un'espressione e che restituisca anche un valore può apparire contorto intuitivo. Tuttavia questo aumenta la regolarità e la semplicità delle regole sintattiche e semantiche del linguaggio.

L'espressione $\langle var \rangle \langle op_2 \rangle = \langle expr_2 \rangle$ è una forma contratta equivalente a $\langle var \rangle = \langle var \rangle \langle op_2 \rangle \langle expr_2 \rangle$. Ad esempio $X+=10$ equivale a $X=X+10$.

- Parentesi (), virgola , e operatore ternario ?:, permettono di raccogliere, concatenare e sequenzializzare più espressioni.

L'espressione in parentesi ($\langle expr \rangle$) produce i side effects di $\langle expr \rangle$ e restituisce il valore e il tipo restituiti da $\langle expr \rangle$. È una parentesi e serve a controllare le priorità di composizione delle operazioni.

L'espressione concatenata $\langle expr_1 \rangle, \langle expr_2 \rangle$ produce i side effects di $\langle expr_1 \rangle$ e poi quelli di $\langle expr_2 \rangle$. Il valore e il tipo restituito sono quelli di $\langle expr_1 \rangle$:

$$\Gamma(\langle expr_1 \rangle, \langle expr_2 \rangle) = \Gamma(\langle expr_1 \rangle)$$

L'espressione ternaria $\langle expr_1 \rangle ? \langle expr_2 \rangle : \langle expr_3 \rangle$ valuta inizialmente $\langle expr_1 \rangle$, producendone i side-effects; se il risultato restituito è vero (ossia se il risultato è diverso da 0) allora esegue $\langle expr_2 \rangle$, altrimenti esegue $\langle expr_3 \rangle$. Il valore e il tipo restituiti sono quelli dell'espressione selezionata tra $\langle expr_2 \rangle$ e $\langle expr_3 \rangle$:

$$\Gamma(\langle expr_1 \rangle ? \langle expr_2 \rangle : \langle expr_3 \rangle) = \begin{cases} \Gamma(\langle expr_2 \rangle) & \text{se } \Gamma(\langle expr_1 \rangle) \neq 0 \\ \Gamma(\langle expr_3 \rangle) & \text{se } \Gamma(\langle expr_1 \rangle) = 0 \end{cases}$$

L'uso congiunto degli operatori di concatenazione e dell'operatore ternario permette di creare espressioni di elevata complessità, che possono eseguire pezzi di codice altrimenti realizzati concatenando istruzioni e controllandone il flusso con istruzioni condizionali. Il vantaggio è che tutto questo è eseguito in-linea, in un'espressione che può apparire nella clausola di guardia di qualche istruzione condizionale o di iterazione. Lo svantaggio è la facilità di errore nella programmazione, e soprattutto la difficoltà nella manutenzione del codice.

Un raro esempio in cui l'operatore ternario può risultare effettivamente conveniente è il caso in cui si voglia esprimere il massimo o il minimo tra due numeri senza doverlo appoggiare su una variabile: l'espressione $\max=x>y?x:y$ assegna alla variabile `max` il maggiore tra `x` e `y`.

- Cast in inglese denota l'azione con cui qualcosa viene messo in una forma, come il cemento armato in una cassaforma. In C, l'espressione di cast (*<type>*)*<expr₁>* produce i side effects dell'espressione *<expr₁>* e restituisce il valore restituito da *<expr₁>* convertito al tipo *<type>*. L'espressione (*<type>* *<expr₁>*) è equivalente a (*<type>*)*<expr₁>*

Il cast può avere un effetto importante, talvolta subdolo, sul risultato restituito da un'espressione. Si consideri il caso di due interi *n* e *m* che valgano rispettivamente 10 e 4 e di un float *x*. L'espressione *x=n/m* assegna a *x* il valore 2 perché la divisione tra *n* e *m* è eseguita con la semantica della divisione tra gli interi. Viceversa, nell'espressione *x=n/({float}m)*, la variabile *m* è convertita al tipo float prima della divisione, e di conseguenza anche *n* viene esteso alla rappresentazione float e la divisione è eseguita con la semantica dei float; il risultato è che in tal caso alla variabile *x* viene assegnato il valore 2.5. Si osservi infine che nell'espressione *x={float} n/m* alla variabile *x* viene assegnato il valore 2 perché la conversione è eseguita quando la divisione ha già restituito il valore 2.

1.5.3 Puntatori

I puntatori sono variabili i cui valori sono indirizzi di locazioni in cui sono memorizzate altre variabili. In linea di principio, avremmo dovuto definirli come parte della sezione 1.5.1 dove abbiamo introdotto le variabili, la loro dichiarazione e il loro riferimento.

Il tipo dei puntatori

Concretamente, il valore di un puntatore è un numero intero che varia tra 0 e $2^n - 1$, dove *n* è il numero di bit che compongono l'indirizzo di una locazione di memoria. In una architettura a 32 bits, un indirizzo è un numero intero compreso tra 0 e $2^{32} - 1$.

A questo punto sarebbe intuitivo pensare che i puntatori potessero essere rappresentati come valori del tipo `unsigned int`.

Questo non sarebbe tuttavia corretto: un tipo è caratterizzato dai valori che rappresenta (e in questo puntatori e interi senza segno coincidono), ma anche dalle operazioni che possono essere applicate ai valori. E sui puntatori si applicano operazioni diverse da quelle che si applicano sugli interi. Nello spazio degli indirizzi di memoria non ha un gran senso eseguire una divisione, o un prodotto, e nemmeno un modulo. Può avere senso una somma o una sottrazione, ma vedremo comunque nel seguito che queste hanno comunque una semantica diversa rispetto a quella applicata agli interi. Ma al di là delle operazioni aritmetiche, di minore rilevanza, quello che fa la differenza tra interi e puntatori è il fatto che sui puntatori è definita

l'operazione che permette di referenziare la variabile identificata dal valore del puntatore.

Il valore del puntatore indica la locazione alla quale è memorizzato il primo byte della variabile puntata, ma non indica su quanti bytes si estende la variabile puntata, né quale sia il formato della sua codifica. D'altra parte, tale informazione è necessaria per potere manipolare la variabile memorizzata a partire dalla locazione indirizzata dal puntatore.

Per fornire al compilatore questa informazione, la dichiarazione di una variabile puntatore, specifica il tipo della variabile puntata. Questo si ottiene introducendo un *modificatore* * prima del nome della variabile. Ad esempio, `int * x_ptr;` dichiara che `x_ptr` è un puntatore a `int`.

La dichiarazione dei puntatori ha una flessibilità che si estende ben oltre quella illustrata nel caso `int * x_ptr;`. Ad esempio è possibile dichiarare che `x_ptr_ptr` è un puntatore a puntatore a intero scrivendo `int ** x_ptr_ptr;`, ed è anche possibile combinare i puntatori con altre estensioni che vedremo nel seguito per array, funzioni e strutture.

Per dare conto della generalità dei casi possibili, torniamo sul formato della dichiarazione che abbiamo dato in sezione 1.5.1, ed estendiamo la definizione della categoria *<decl>* con l'aggiunta di un modificatore prefisso *:

```
<declaration> ::= <type> <decl>;
<decl> ::= <identifier> | * <decl> | ...
```

La semantica della dichiarazione `<type> * <decl>;` è definita in riferimento alla semantica della dichiarazione `<type> <decl>;`. E questa è la ragione per cui si dice che il simbolo * agisce come *modificatore della dichiarazione*. Specificamente, `<type> * <decl>;` dichiara una variabile puntatore a una variabile del tipo che sarebbe dichiarato da `<type> <decl>;`, e le associa il nome che sarebbe associato alla variabile dichiarata da `<type> <decl>;`.

In particolare, nel caso più comune di una dichiarazione nel formato `<type> * <identifier>;` (ad esempio questo è il caso della dichiarazione `int * x_ptr;`), la dichiarazione si legge con la dizione `<identifier>` è un puntatore a una variabile di tipo `int`.

Si noti che la struttura ricorsiva della definizione di *<decl>* permette la presenza di un numero arbitrario di simboli * prima del termine *<identifier>* che definisce il nome dalla variabile. Questo è per esempio il caso della dichiarazione `int ** x_ptr_ptr;` che può essere ridotto al simbolo iniziale *<declaration>* applicando per due volte la riduzione *<decl> → * <decl>* e per una volta la riduzione terminale *<decl> → <identifier>* (vedi Fig.1.13). Il significato associato all'espressione è allora `x_ptr_ptr` è un puntatore a una variabile di tipo puntatore a una variabile di tipo puntatore. In breve `x_ptr_ptr` è un puntatore a un puntatore a intero.

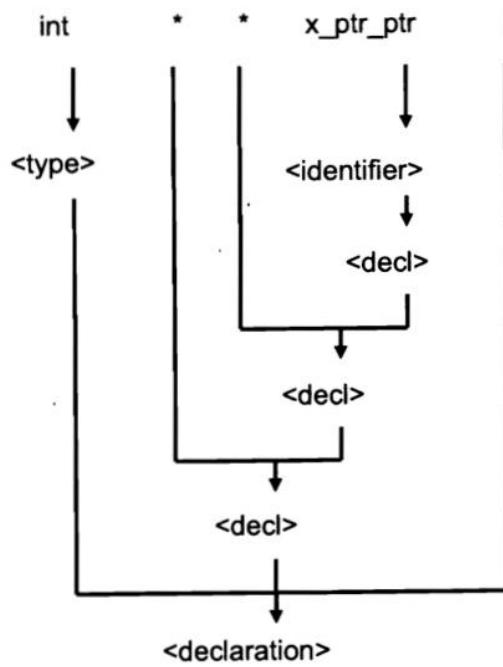


Figura 1.13: L'albero sintattico che riconosce una dichiarazione di variabile nella forma `int ** x_ptr_ptr;`.

I puntatori nelle espressioni

I puntatori sono variabili, nel senso che il loro valore può cambiare nel corso della computazione. Dal punto di vista sintattico appartengono alla categoria `<var>`, e possono quindi apparire all'interno di una espressione nelle posizioni in cui può apparire una qualsiasi variabile.

Esistono alcune limitazioni e particolarità nell'uso e significato degli operatori. In particolare è rilevante il caso delle operazioni aritmetiche sui puntatori.

Se `x_ptr` è un puntatore a una variabile di tipo `<type>`, allora `x_ptr+2` restituisce un valore, ancora di tipo puntatore a `<type>`, che è pari al valore restituito da `x_ptr` con l'aggiunta di un incremento; questo incremento non è 2 ma `2*sizeof(<type>)`, dove `sizeof(<type>)` denota il numero di bytes impiegati per la rappresentazione di una variabile del tipo `<type>`. Per effetto di questo, `x_ptr+2` restituisce l'indirizzo due variabili di tipo `<type>` avanti a partire da `x_ptr`. Generalizzando, la somma tra un puntatore e un numero intero restituisce l'indirizzo del puntatore incrementato del numero intero moltiplicato per la dimensione in byte del tipo puntato. La stessa semantica si applica ovviamente all'operazione di differenza tra un puntatore e un intero, e alle operazioni di incremento e decremento. In maniera consistente, la differenza tra due puntatori a variabili dello stesso tipo restituisce un numero intero che esprime il numero

di variabili del tipo che possono essere memorizzate tra le locazioni puntate dai due puntatori. Le operazioni di prodotto, divisione e modulo non si applicano ai puntatori.

Le operazioni aritmetiche sui puntatori sono spesso abusate nella pratica della programmazione c. Nella maggioranza dei casi è possibile evitarne l'uso facendo ricorso agli arrays, che permettono al programmatore di ignorare i dettagli sulla realizzazione fisica dei puntatori e che producono codice più disciplinato, manutenibile e verificabile.

Riferimento a variabili attraverso puntatori

L'utilità pratica dei puntatori è essenzialmente legata alla possibilità di fare riferimento a una variabile attraverso un puntatore che ne contiene l'indirizzo (dereferenziazione di un indirizzo). L'operatore che permette di fare questo è denotato da un * ed è applicato in forma prefissa a una espressione che restituisce un valore del tipo di un puntatore. Ad esempio, se `x_ptr` è un puntatore, allora `*x_ptr` è un riferimento a variabile che denota la variabile puntata da `x_ptr`, e l'espressione `*x_ptr=12` le assegna il valore 12.

Il fatto che `*x_ptr` possa apparire a sinistra di un assegnamento non deve passare inosservato. Al di là del ruolo fondamentale che questo assumerà nell'uso delle funzioni, questo significa che `*x_ptr` non è un'espressione che restituisce il valore della variabile puntata da `x_ptr`, ma è piuttosto un riferimento a variabile.

Dal punto di vista sintattico, questo implica che l'operatore di dereferenziazione * prefisso è parte della clausola che definisce la categoria `<var>`. Estendiamo quella clausola che avevamo lasciato sospesa nella sezione 1.5.1:

`<var> ::= <identifier> | *<expr> | ...`

il cui significato è che, se `<expr>` è un'espressione che restituisce un valore di tipo indirizzo di una variabile di un qualche tipo, allora `*<expr>` denota la variabile di quel tipo che è memorizzata a partire dalla locazione definita dall'indirizzo restituito dall'espressione `<expr>`.

Si osservi che nella clausola sopra, l'operatore * è prefisso a una espressione che restituisce un indirizzo. Come caso particolare, quell'espressione può essere il nome di una variabile puntatore (in base alla clausola `<expr> ::= ... | <var> | ...`); questo è ad esempio il caso del riferimento `*x_ptr`. Più in generale, la clausola sintattica che abbiamo scritto indica che l'operatore * può essere applicato a una qualsiasi espressione che restituisce un indirizzo. Questo potrebbe essere ad esempio il caso del riferimento `*(x_ptr+3)`. Riferimenti ancora più elaborati potranno essere espressi con l'introduzione di arrays e variabili strutturate.

Operatore di indirizzo

L'operatore `&` è all'incirca l'inverso di `*`: se `x` è una variabile, `&x` è un'espressione che restituisce l'indirizzo di `x`.

Dal punto di vista sintattico, l'operatore `&` si colloca nella clausola delle espressioni `<expr>`, che qui estendiamo rispetto alla definizione che abbiamo lasciata aperta in sezione 1.5.2:

```
<expr> ::= ... | &<var> | ...
```

Per quanto attiene alla sua semantica, l'espressione `&<var>` non produce side-effects e restituisce il valore dell'indirizzo della variabile referenziata da `<var>`. Il valore restituito ha ovviamente il tipo di un indirizzo di variabile del tipo di `<var>`.

È utile sottolineare il fatto che `&x` è un'espressione (di tipo indirizzo di variabile del tipo di `x`) e non un riferimento a variabile. Questo significa che `&x` non può apparire a sinistra di un assegnamento, il che del resto implicherebbe una modifica all'indirizzo a cui è locata una variabile, cosa che non è compatibile con la definizione stessa di variabile.

Gli operatori `*` e `&` sono spesso associati nella descrizione del linguaggio C, per dare conto del loro ruolo complementare e per il fatto che i due operatori sono naturalmente impiegati congiuntamente. Non devono però sfuggire le sostanziali differenze tra i due: `*` serve a denotare una variabile mentre `&` serve solo a restituire un valore; l'implementazione di `*` richiede che il compilatore conosca il tipo della variabile a cui `*` è applicato (e questa è la ragione ultima per cui i puntatori hanno un tipo), mentre `&` ha la stessa implementazione qualsiasi sia la variabile a cui è applicato.

I puntatori a void

Nella dichiarazione di un puntatore, il tipo delle variabili puntate può essere specificato come `void`. In questo caso non è determinato il tipo della variabile di cui il puntatore conterrà l'indirizzo.

In questa condizione, il compilatore è comunque in grado di interpretare la dichiarazione allocando i bytes che sono necessari alla rappresentazione del valore del puntatore, visto che in ultimo la dimensione e il formato di un indirizzo non dipendono dal tipo della variabile puntata. Il compilatore è anche in grado di compilare operazioni di assegnamento nelle quali al puntatore viene assegnato il valore di un indirizzo (di qualsiasi tipo).

Tuttavia, se il compilatore non conosce il tipo della variabile puntata non è in grado di tradurre l'operatore `*`. Perché, fino a che non sia specificato il tipo della variabile puntata, il compilatore conosce l'indirizzo a partire dal quale

è memorizzata la variabile puntata, ma non conosce su quanti bytes si estende la sua rappresentazione, né quale sia la sua codifica. Per potere applicare l'operatore `*` occorre allora che nell'espressione del riferimento si specifichi il tipo di variabile al quale si ritiene di applicare il puntatore. Questo si ottiene con una operazione di cast. Chiariamo il concetto con un esempio.

```
void * ptr;

int n;
int m;
float x;
float y;
int select;

...
if(select==0)
    ptr=&n;
else
    ptr=&x;
...
if(select==0)
    m=*((int *)ptr);
else
    y=*((float *)ptr);
...
```

La variabile `ptr` è dichiarata come puntatore a `void`. Il compilatore alloca i bytes necessari per rappresentare un indirizzo: sono gli stessi indipendentemente dal tipo della variabile che è memorizzata a partire dall'indirizzo. Per la stessa ragione il compilatore non ha neppure problemi ad applicare l'operatore `&` alle variabili `n` e `x` e ad assegnare il valore risultante a `ptr`. Viceversa, l'estrazione della variabile a cui `ptr` punta attraverso l'operatore `*` richiede che il compilatore sappia il tipo della variabile a cui si intende che `ptr` stia puntando. Questo si ottiene con il cast `(int *)` se si intende che il puntatore indirizzi un intero, oppure `(float *)` se si intende che il puntatore indirizzi un float.

L'esempio illustra come l'uso congiunto del puntatore a `void` con il cast permette di ritardare fino al momento del riferimento la determinazione del tipo della variabile indirizzata da un puntatore, permettendo l'uso di puntatori che in momenti diversi dell'esecuzione puntano a variabili di tipo diverso.

L'esempio che abbiamo riportato è corretto. Ma non è contemplato nella sintassi che abbiamo fino qui definito per l'operazione di cast, la quale permette che il tipo su cui si fa il cast sia un tipo elementare ma non un puntatore. Il limite è superato completando la definizione della categoria `<cast-type>` che avevamo lasciato in sospeso nel capitolo 1.5.2:

```
<expr> ::= ... | (<cast-type>) <expr> | ...
<cast-type> ::= <type> | <cast-type> *
```

È utile osservare che anche con questa aggiunta, il tipo `<cast-type>` non è sufficiente a specificare tutti i possibili tipi con i quali è possibile dichiarare una variabile. Questo implica una limitazione nell'uso di puntatori void che non permette di associare a un puntatore void qualunque tipo. Ad esempio non è possibile usare un puntatore a void per referenziare una funzione. Si tratta peraltro di limitazioni che è difficile arrivare a osservare, anche nella programmazione avanzata.

A questo punto è possibile tornare indietro alla definizione di tipi elementari del C dove avevamo lasciato in sospeso la questione del tipo void dicendo: Il tipo void e il tipo nullo. Permette di soddisfare una sintassi che richiede la presenza di uno specificatore di tipo senza però indicare quale sarà questo tipo. In sostanza è qualcosa che gioca il ruolo di un tipo dal punto di vista sintattico ma non da quello semantico. Quella affermazione dovrebbe ora essere chiara.

I puntatori void sono un oggetto abbastanza raffinato e potente della programmazione C. Esistono dei patterns di uso comune nei quali possono essere risolutivi, ad esempio nella realizzazione di un iteratore che esamina un insieme di dati di tipo diverso. È comunque difficile averne bisogno prima di avere raggiunto un livello di uso molto avanzato del linguaggio.

Esempi

Illustriamo alcuni modi di uso dei puntatori. Alcuni corrispondono a patterns di uso comune. Altri sono casi estremi riportati per illustrare dettagli delle regole sintattiche e semantiche.

Il primo esempio illustra l'uso congiunto degli operatori * e &:

```
int x;
int * x_ptr;

x=10;      // x vale 10
x_ptr=&x;    // x_ptr punta a x
*x_ptr=12; // x vale 12
```

Nel frammento di codice sono dichiarate due variabili: `x` è un intero, `x_ptr` è un puntatore a intero. Inizialmente le due variabili contengono valori non prevedibili al programmatore. L'espressione `x=10` assegna il valore 10 alla variabile `x`. L'espressione `x_ptr=&x` assegna a `x_ptr` il valore dell'indirizzo di `x`. Dunque al momento dell'ultima istruzione `*x_ptr` denota la variabile `x`. L'effetto dell'assegnazione `*x_ptr=12` è dunque quello di assegnare il valore 12 alla variabile `x`.

Il prossimo esempio illustra un uso limite dei puntatori, che ha la funzione di uno scioglilingua più che rispondere a una effettiva utilità.

```
float x;
float y;
float z;
float * x_ptr;

x_ptr=&x;      // x_ptr punta a x
*(x_ptr+2)=12; // z vale 12
```

Nel leggere l'esempio occorre anticipare che le variabili di un programma sono allocate dal compilatore in posizioni contigue della memoria (ne ripareremo nella trattazione delle funzioni). Dunque se la variabile *x* si trova all'indirizzo *address*, allora *y* e *z* si trovano all'indirizzo *address+4* e *address+8* (essendo 4 il numero di bytes impiegati per codificare un float).

L'espressione *x_ptr=&x* assegna a *x_ptr* il puntatore a *x*. L'espressione *x_ptr+2* restituisce l'indirizzo di *z* che si trova due variabili float avanti a partire da *x* (essendo *x_ptr* un puntatore a float che punta *x*). Dunque **(x_ptr+2)* denota la variabile *z*, e **(x_ptr+2)=12* assegna il valore 12 alla variabile *z*.

Vediamo infine un esempio che illustra l'uso di un puntatore a puntatore, che spesso in questo testo chiameremo doppio puntatore:

```
int x;           // x e' un intero
int * x_ptr;     // x_ptr e' un puntatore a intero
int ** x_ptr_ptr; // x_ptr_ptr e' un punt. a punt. a intero

x_ptr=&x;      // x_ptr punta a x
x_ptr_ptr=&x_ptr; // x_ptr_ptr punta a x_ptr
**x_ptr_ptr=12; // x vale 12
```

L'assegnamento ***x_ptr_ptr=12* assegna il valore 12 alla variabile *x*. Si noti che non sarebbe stato legale inizializzare *x_ptr_ptr* usando applicando due volte l'operatore & nell'espressione *x_ptr_ptr=&&x*. Infatti *&x* è un'espressione e l'operatore & può essere prefisso a una variabile ma non a un'espressione.

Il doppio puntatore è usato frequentemente nella programmazione c, e vedremo che è necessario per costruire funzioni che possano modificare un puntatore. Che è un caso frequente nel trattamento di molte strutture dati rilevanti.

È utile osservare che il doppio puntatore non è una categoria particolare del linguaggio, ma piuttosto una particolare inflessione della regola generale che definisce i puntatori a variabili di qualunque tipo. Vedremo nel seguito che applicando la regola nella sua generalità è possibile definire puntatori a qualunque cosa: puntatori ad array di interi, puntatori ad array di puntatori a interi, puntatori ad array di puntatori a funzioni.

1.5.4 Array

Un array è un insieme di variabili, dello stesso tipo, che possono essere referenziate attraverso un nome collettivo e un indice che le distingue tra loro.

Nella prospettiva fisica del compilatore, l'array è realizzato come una sequenza di locazioni contigue. Questo permette di identificare la locazione a cui è posta ciascuna variabile della sequenza attraverso un indirizzo di base e un offset.

Come già osservato per i puntatori, anche gli array in linea di principio dovrebbero essere collocati assieme alle variabili nel capitolo 1.5.1. Anche per gli array, si tratterà comunque di precisare come sono espresse la dichiarazione e il riferimento.

Dichiarazione di variabili array

Per dichiarare un array occorre stabilire il tipo e il numero delle variabili che lo compongono, e il nome con cui il programma può fare riferimento all'array. Per questo viene estesa la categoria sintattica *<decl>* introdotta nel capitolo 1.5.1 con l'aggiunta di un *modificatore suffisso* [*<const>*]:

```
<declaration> ::= <type> <decl>;  
<decl> ::= <identifier> | * <decl> | <decl> [<const>] | ...
```

La semantica della dichiarazione *<type> <decl> [<const>]*; è definita in relazione alla semantica della dichiarazione *<type> <decl>*; , ragione per cui la parentesi quadra viene detta modificatore [*<const>*] . Specificamente: *<type> <decl> [<const>]*; dichiara una variabile array di *<const>* variabili del tipo delle variabili che sarebbero dichiarate da *<type> <decl>*; e all'array viene associato il nome che sarebbe associato alla variabile dichiarata da *<type> <decl>* .

Ad esempio, la dichiarazione `float X[128];` dichiara una variabile array costituita da 128 variabili di tipo float (ovvero del tipo della variabile che sarebbe dichiarata da `float X;`) e le associa il nome X (ovvero il nome che sarebbe associato alla variabile dichiarata da `float X;`).

Il compilatore interpreta la dichiarazione riservando un segmento di memoria sufficiente a rappresentare le variabili dell'array in locazioni contigue. Evidentemente, l'ampiezza di tale segmento è pari al numero *<const>* moltiplicato per il numero di bytes richiesti dalla codifica di una variabile del tipo specificato da *<type> <decl>*.

È utile osservare che la clausola che definisce *<decl>* ha una struttura ricorsiva che permette di associare a un *<identifier>* un numero arbitrario modificatori prefissi * e suffissi [*<const>*]. Questo introduce una ambiguità sull'ordine con cui devono essere considerati i modificatori * e [*<const>*]. Illustriamo il problema nel riferimento dell'esempio di alcune dichiarazioni:

```

float X;           // X e' un float
float * X_ptr;    // X_ptr e' un puntatore a float
float X_array[128]; // X_array e' un array di 128 variabili float
float * XX[128];  // XX e' un array di 128 puntatori a intero;

```

L'ambiguità è risolta definendo una priorità nell'ordine con cui sono interpretati i modificatori suffissi e prefissi nella semantica di una dichiarazione di variabile: il modificatore suffisso [*const*] ha priorità sul modificatore prefisso *. Per effetto della priorità, la variabile dichiarata da `float * XX[128];` è un array di 128 puntatori a float e non un puntatore ad array di 128 float.

Dichiarazioni complesse

Con la combinazione di puntatori ed array, l'interpretazione della dichiarazione di una variabile del c può diventare abbastanza complessa per il programmatore. Ad esempio, può capitare di dovere determinare quale è il tipo di una variabile dichiarata con `float **XX[12][18];`. E può esserci di peggio se si combinano in tutta la loro espressività puntatori, array e, nel seguito funzioni.

In linea di principio, il metodo che determinare in maniera inequivocabile il significato di una dichiarazione è quello di applicare successivamente il significato dei modificatori, riducendo la interpretazione di una dichiarazione complessa a quella delle dichiarazioni che essa modifica. Ad esempio: il tipo della variabile dichiarata da `float * XX[128];` è un array di 128 variabili del tipo della variabile che sarebbe dichiarata da `float * XX;`; che a sua volta è un puntatore a una variabile del tipo che sarebbe dichiarato da `float XX;`, ovvero un float. Alla fine del palo, XX è un array di 128 puntatori a variabili di tipo float: Fig.1.14 mostra l'albero sintattico che riduce l'espressione alla forma di una dichiarazione.

Il metodo, di per sé abbastanza macchinoso, è efficacemente sintetizzato in una regola del pollice che permette di interpretare agevolmente il significato di dichiarazioni anche complesse: la dichiarazione si legge a partire dal nome che identifica la variabile, espandendo prima verso destra fintanto che ci sono modificatori suffissi, e poi verso sinistra fino ad arrivare al tipo elementare; nella lettura, al nome della variabile si fa seguire la dizione ... è *un* ..., a ogni suffisso della forma [*const*] si fa seguire la dizione ... *array di* <*const*> *variabili di tipo* ..., a ogni prefisso della forma * si fa seguire la dizione ... *puntatore a una variabile di tipo* Ad esempio, `float **XX[12][18];` viene letto in XX è *un - array di 12 variabili di tipo - array di 18 variabili di tipo - puntatore a variabile di tipo - puntatore a variabile di tipo - int*, o più brevemente: XX è *un array di 12 array di 18 puntatori a puntatore a int*. Fig.1.14 illustra il concetto.

È utile osservare che, al di là del formato un po' cialtrone con cui è espressa, questa regola riflette esattamente la semantica associata ai singoli termini che appaiono nella sintassi delle categorie <*declaration*> e <*decl*>.

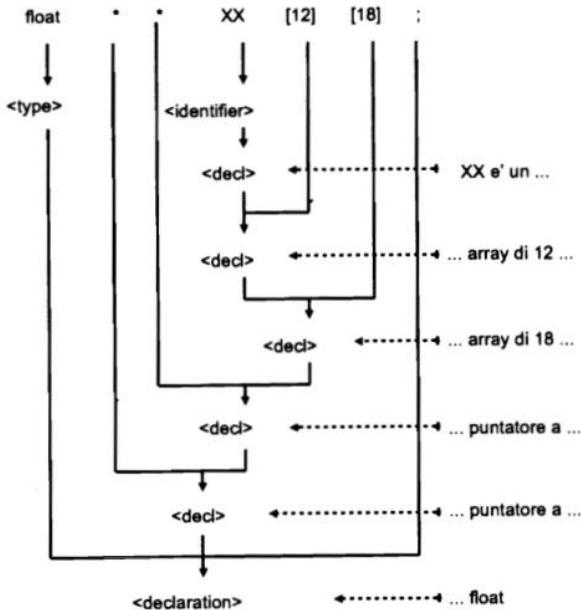


Figura 1.14: L'albero sintattico per la dichiarazione `float ** XX[12][18];`. Il modificatore suffisso `[const]` ha precedenza sul modificatore prefisso `*`. Le annotazioni sulla destra mostrano l'interpretazione della dichiarazione,

La regola che abbiamo riportato semplifica sostanzialmente il problema di leggere una dichiarazione. Per il problema inverso, quello di scrivere la dichiarazione che associa a una variabile un tipo assegnato, il procedimento è sostanzialmente lo stesso, all'inverso: si parte scrivendo il nome, si procede estendendo l'espressione verso destra fino a che si incontrano termini del tipo array di, poi si estende a sinistra fino a che si incontrano termini del tipo puntatore a, e infine si applica il tipo elementare. Ad esempio, se il problema è quello di scrivere in c la dichiarazione `XX` è un array di 18 puntatori a `int`, la sequenza con cui si ottiene la dichiarazione è:

```

...XX...      // XX ...
...XX[18]... // XX e' un array di 18 ...
...* XX[18]; // XX e' un array di 18 puntatori a ...
int * XX[18]; // XX e' un array di 18 puntatori a intero

```

Applicando il metodo si scopre che non tutte le dichiarazioni possono essere realizzate con la sintassi che abbiamo fino qui definito. Ad esempio, non è possibile la dichiarazione `ptr` è un puntatore ad array di 128 interi. Apparentemente questo potrebbe essere `int * ptr[128];`, che però in base alla regola che abbiamo dato si legge come array di 128 puntatori a intero e non come puntatore ad array di 128 interi. Analiticamente, questo dipende dall'avere dato priorità ai modificatori suffissi `[]` che impedisce di avere una dichiarazione in cui si dichiara il puntatore ad un array.

Il limite è superato estendendo ulteriormente la sintassi della categoria *<decl>* con l'introduzione di parentesi tonde che permettono di forzare la priorità nella lettura della dichiarazione:

```
<declaration> ::= <type> <decl>;  
<decl> ::= ... | (<decl>) | ...
```

Anche in questo caso, la semantica della dichiarazione *<type> (<decl>)*; è definita come modifica a quella della dichiarazione *<type> <decl>;*. La cosa è semplice perché *<type> (<decl>)*; definisce una variabile che ha lo stesso nome e lo stesso tipo della variabile definita da *<type> <decl>;*. L'effetto della parentesi non è sulla semantica ma sul controllo della priorità con cui sono applicati i modificatori: nella lettura delle dichiarazioni, la parentesi impone di dare precedenza ai modificatori interni alla parentesi rispetto a quelli esterni. Nella scrittura, si aggiunge una parentesi tutte le volte che occorre trascrivere un termine del tipo puntatore a che sarà poi seguito da un termine array di.

La dichiarazione *ptr* è un puntatore ad array di 128 interi (che di per sé non serve a molto nella programmazione) può adesso essere espressa nella forma:

```
...ptr...      // ptr ...  
...(* ptr)... // ptr e' un puntatore a ...  
...(* ptr)[128]; // ptr e' un puntatore ad array di 128 ...  
int (* ptr)[128]; // ptr e' un puntatore ad array di 128 interi
```

Riferimento a variabili array

Il nome associato a una variabile array denota il valore (costante) dell'indirizzo a partire dal quale l'array è memorizzato. Il tipo di tale valore è quello di un puntatore a variabili del tipo raccolte nell'array.

Le singole variabili che costituiscono l'array possono essere referenziate combinando il nome con un indice (un numero intero non negativo) che numera gli elementi dell'array a partire da 0. La combinazione è espressa specificando l'indice entro parentesi quadre applicate in forma suffissa al nome dell'array.

Per esempio consideriamo il caso di un array *XX* dichiarato come *int X[128];*. Il nome *XX* denota una costante di tipo indirizzo di *int* il cui valore è l'indirizzo della locazione a partire dalla quale è memorizzato l'array; il riferimento *XX[0]* denota la prima variabile dell'array, *XX[1]* la seconda, e *XX[127]* l'ultima.

Più in generale, la forma del riferimento permette che l'indice sia ottenuto come valore restituito da una espressione (intera), mentre il riferimento all'array può essere espresso come valore di una espressione che restituisce un puntatore a variabile del tipo delle variabili nell'array. Questo è specificato estendendo la categoria sintattica *<var>*:

*<var> ::= <identifier> | *<expr> | <expr>[<expr>] | ...*

dove si assume (è un vincolo contestuale) che le espressioni *<expr>* e *<expr>* restituiscono valori di tipo intero e di tipo indirizzo, rispettivamente.

Per quanto attiene alla semantica, se *ptr* e *n* sono i valori restituiti dalle espressioni *<expr>* e *<expr>*, e se *t* è il tipo di variabile a cui punta il valore *ptr*, allora il riferimento *<expr>[<expr>]* denota una variabile di tipo *t* che si trova all'indirizzo *ptr + n*. Dove la somma *ptr + n* è intesa con la semantica della somma sui puntatori, di modo che *ptr + n* è l'indirizzo a cui si trova la *n + 1*-esima variabile di tipo *t* a partire dall'indirizzo *ptr*.

Traspare da questa semantica il fatto che l'operatore di selezione [] è sostanzialmente equivalente a una combinazione dell'operatore aritmetico di somma sui puntatori con l'operatore di riferimento *. In effetti, i riferimenti *<expr>[<expr>]* e **(<expr>+<expr>)* sono del tutto equivalenti. Si capisce in questa prospettiva perché la numerazione dell'indice degli array parte da 0 e non da 1.

Vediamo un esempio, poco significativo ai fini pratici, ma adeguato a illustrare i concetti introdotti:

```
float X[128];
float * ptr;
short int count;

ptr=X;           // ptr punta la prima locazione dell'array X
ptr[0]=10        // assegna 10 alla prima variabile di X
(ptr+1)[0]=20;   // assegna 20 alla seconda variabile di X
count=1;
(ptr+1)[count+1]=40; // assegna 40 alla quarta variabile di X
X[200]=0;        // scrive uno 0 in formato float
                  // alla locazione che si trova partendo
                  // dall'indirizzo di base dell'array X
                  // e avanzando di 200 variabili float (!)
```

Fig.1.15 illustra l'allocazione in memoria delle tre variabili *X*, *ptr* e *count*. Per *X* viene riservato lo spazio di 128 variabili float (ovvero 128×4 bytes), per *ptr* sono riservati 4 bytes (indirizzi a 32 bit), mentre per *count* viene riservato 1 byte. I nomi *ptr* e *count* denotano le due variabili, il cui valore iniziale è impredicibile, mentre *X* denota una costante il cui valore è l'indirizzo a partire dal quale è stato riservato il segmento per la rappresentazione dell'array.

L'ultimo assegnamento spara a caso in memoria, visto che l'array *X* è dichiarato di lunghezza 128 e non contiene 201 locazioni. Tuttavia il compilatore non segnala il problema. Né potrebbe segnalarlo, visto che in generale l'indice che sta tra le parentesi quadre di un riferimento può essere una espressione che restituisce un valore dipendente da valori definiti in run-time e non determinati al tempo della compilazione. Potrebbe esistere un controllo al tempo di esecuzione introducendo

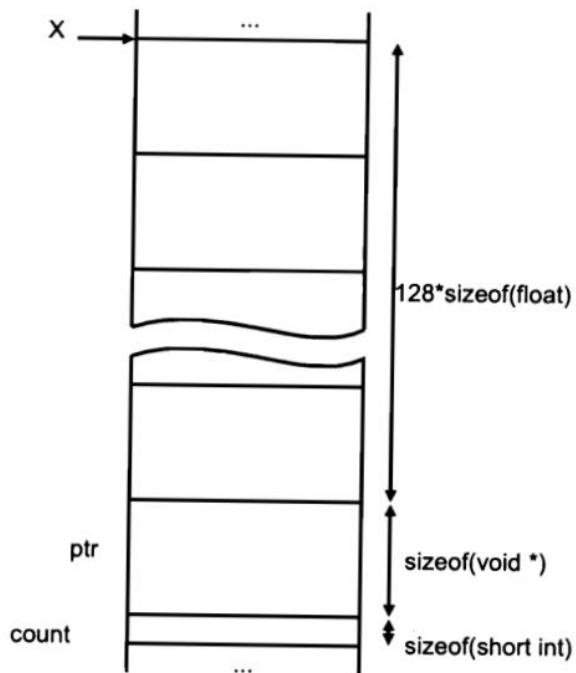


Figura 1.15: La allocazione in memoria di tre variabili dichiarate come: `float x[128];`, `float * ptr;` e `short int count;`. Il simbolo `X` denota l'indirizzo a partire dal quale è riservato lo spazio per le 128 variabili nell'array. Viceversa `ptr` e `count` sono nomi di variabili.

in fase di compilazione del codice aggiuntivo che comparasse dinamicamente il valore dell'indice del riferimento contro la dimensione statica dell'array e che generasse una eccezione in caso di un riferimento fuori dal range. Una soluzione di questo tipo ridurrebbe la possibilità di errori nella programmazione e migliorerebbe la sicurezza nella gestione della memoria. Tuttavia ridurrebbe anche l'efficienza del programma e la flessibilità nell'uso del linguaggio.

Array multi-dimensionali

Il modificatore di array permette di raccogliere variabili di qualunque tipo. In particolare è possibile dichiarare un array di variabili che sono a loro volta di tipo array. Questo permette di rappresentare in maniera naturale oggetti quali matrici e tensori.

Consideriamo il caso in cui siano assegnate due matrici di numero reali A e B e se ne voglia calcolare il prodotto C . Come noto dall'algebra lineare, se A e B hanno dimensione $I \times H$ e $H \times J$, allora C ha dimensione $I \times J$ e il suo elemento in posizione ij è il risultato del prodotto scalare tra la riga i -esima di A e la colonna j -esima di B :

$$A_{I \times H} B_{H \times J} = C_{I \times J}$$

$$c_{ij} = \sum_{h=1, H} A_{ih} B_{hj}$$

Le tre matrici possono essere rappresentate come tre array di array di variabili float, che denominiamo A, B, e C:

```
float A[32][16];
float B[16][4];
float C[32][4];
int i;
int j;
int h;

for(i=0;i<32;i++)
    for(j=0;j<4;j++)
    {
        C[i][j]=0;
        for(h=0;h<16;h++)
            C[i][j]+=A[i][h]*B[h][j];
    }
```

È utile soffermarsi a riflettere sul modo con cui vengono effettivamente rappresentate in memoria le tre matrici A, B, e C. Che nella prospettiva della geometria lineare sono oggetti di natura bi-dimensionale, ma che comunque devono essere rappresentate nella memoria di un elaboratore che è organizzata come una sequenza lineare di locazioni.

Consideriamo il caso di C, anche illustrato in Fig.1.16: la variabile è dichiarata come array di 32 array di 4 float; dunque C denota un valore che punta a un segmento di memoria nel quale sono rappresentate 32 variabili; ciascuna di queste 32 variabili è a sua volta un segmento di 4 variabili float. Di conseguenza, le singole variabili dell'array sono serializzate nella rappresentazione di memoria secondo un ordine riga-colonna: nelle prime locazioni abbiamo gli elementi della prima riga, poi quelli della seconda e così via. Più precisamente l'elemento selezionato dagli indici [i] [j] si trova nella posizione $i*J+j$, dove J denota il numero di colonne di C. Si osservi che questo non è una convenzione che aggiungiamo ora, ma discende invece da sintassi e semantica della dichiarazione di variabili *<declaration>*, e in particolare dalla loro applicazione al caso di un array di array di float.

La dichiarazione di array di array rende del tutto naturale la manipolazione degli array nella rappresentazione di matrici. E questo può ovviamente essere esteso al caso di matrici a 3 o più dimensioni.

Tuttavia, l'uso di array di array incontra grosse limitazioni nell'uso di funzioni e di variabili allocate dinamicamente. Al di là di quello che può essere intuito a questo punto, questo ha a che fare sia con il modo con cui le funzioni possono condividere

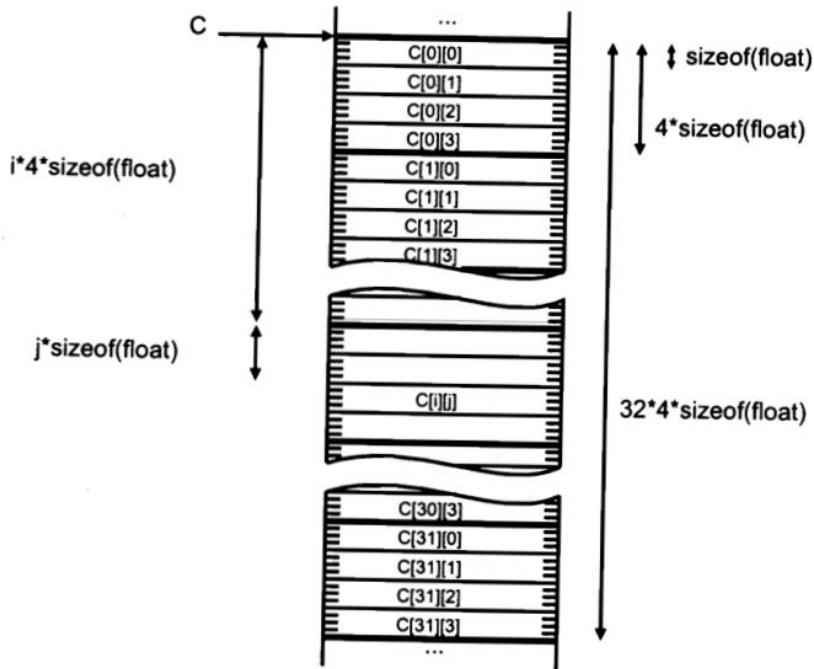


Figura 1.16: Allocazione in memoria di un array dichiarato come `float C[32][4];` Il nome `C` denota l'indirizzo di base dell'array. La variabile referenziata come `C[i][j]` si trova nella locazione `C+i*4+j`.

variabili che con le limitazioni sull'espressione del tipo su cui è possibile operare un cast.

Nella pratica della programmazione C, le matrici a più dimensioni sono comunemente dichiarate come array monodimensionali, lasciando al programmatore la gestione della indicizzazione. In questo caso, una matrice di dimensioni $I \times J$ è dichiarata come un array monodimensionale di dimensione $I * J$ e l'elemento $[i][j]$ è selezionato con l'indice $[i * J + j]$.

In questo approccio il prodotto delle matrici dell'esempio viene riscritto come:

```

float A[32*16];
float B[16*4];
float C[32*4];
int i;
int j;
int h;

for(i=0;i<32;i++)
    for(j=0;j<4;j++)
    {
        C[i*4+j]=0;
        for(h=0;h<16;h++)
            C[i*3+j]+=A[i*16+h]*B[h*3+j];
    }
}

```

È utile osservare che il metodo può essere applicato a matrici a 3 o più dimensioni. Nel caso di una matrice a tre dimensioni, la dichiarazione con più indici avrebbe la forma float X3[128][16][4]; e l'elemento di indice ijk verrebbe referenziato come $X3[i][j][k]$. La dichiarazione a un indice unico viene invece scritta come float X3[128*16*4];, e l'elemento di indice ijk viene referenziato come $X3[i*J+j*K+k]$, dove J e K denotano il numero di valori del secondo e del terzo indice, 16 e 4 nel nostro esempio. Si osservi che, ai fini del riferimento è irrilevante il numero di valori del primo indice, 128 nel nostro esempio.

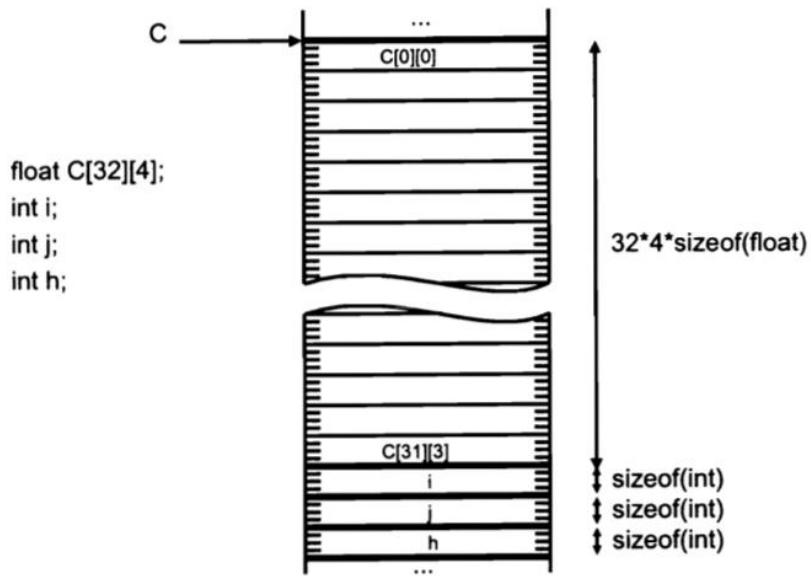


Figura 1.17: Allocazione in memoria 4 variabili dichiarate in successione come `float C[32][4]; int i; int j; int h;`

Allocazione dinamica

Nella dichiarazione di un array la specifica del numero di elementi è necessaria al compilatore per determinare l'ampiezza del segmento di memoria che deve essere riservato. Ad esempio, facendo ancora riferimento al frammento di codice discusso poco sopra che calcola il prodotto di due matrici, il compilatore riserva all'array C un segmento di $32*4*sizeof(\text{float})$ bytes, e colloca nei bytes che seguono le variabili intere i , j e h (vedi Fig.1.17). Evidentemente, per riservare spazio alle variabili i , j e h , il compilatore deve avere determinata la dimensione di C . Si capisce da questo che non è possibile dichiarare un array di dimensione variabile, il cui valore sia determinato al tempo di esecuzione, magari in base a un dato ricevuto in input durante l'esecuzione del programma.

Una volta chiarito che questa limitazione risponde a una necessità oggettiva, non superabile, del compilatore, occorre tuttavia osservare che essa può diventare

un problema concreto per molti programmi. Ad esempio si consideri il frammento di codice dell'esempio precedente che calcola il prodotto di due matrici A e B, di dimensione 128×16 e 16×3 , rispettivamente. Se volessimo calcolare il prodotto di due matrici di dimensione 7×4 e 4×3 dovremmo sostituire tutte le occorrenze di 128, 16 e 4 con 7, 4 e 3. Facendo attenzione a non confondersi con altre costanti del codice, magari uguali a 128, 16 o 4 per ragioni del tutto accidentali.

La difficoltà è alleviata dall'uso della direttiva di preprocessing `#define` che permette di denotare un valore con un simbolo. Con questa possiamo riscrivere il calcolo del prodotto delle due matrici nella forma:

```
#define I 32
#define H 16
#define J 4

float A[I*H];
float B[H*J];
float C[I*J];
int i;
int j;
int h;

for(i=0;i<I;i++)
    for(j=0;j<J;j++)
    {
        C[i*J+j]=0;
        for(h=0;h<H;h++)
            C[i*J+j]+=A[i*H+h]*B[h*J+j];
    }
```

I simboli I, H, e J denotano i valori 32, 16 e 4, rispettivamente. Prima dell'avvio del compilatore, il testo del codice viene pre-processato in modo da sostituire tutte le occorrenze dei simboli I, H, e J con i rispettivi valori.

Così facendo, il codice che calcola il prodotto di due matrici di dimensione 7×4 e 4×3 può essere ottenuto modificando le sole linee `#define`:

```
#define I 7
#define H 4
#define J 3
...
```

L'uso della direttiva `#define` riduce la complessità di editing del programma e la possibilità di commettere errori. Tuttavia non permette di trattare arrays la cui dimensione viene determinata durante l'esecuzione: i valori 7, 4, e 3 sono comunque fissati prima che il programma venga lanciato.

Per risolvere il problema occorre che lo spazio per la rappresentazione dell'array sia allocato nel corso dell'esecuzione del programma. Questo non lo può fare il

compilatore, ma può essere realizzato interfacciando il programma con il sistema operativo che gestisce le risorse della macchina. Tra le quali la memoria.

La funzione di libreria `malloc()` richiede al sistema operativo l'allocazione di un array di variabili `char` di dimensione determinata al momento della chiamata e restituisce il puntatore alla locazione della memoria a partire dalla quale è stato riservato lo spazio per l'array.

Usando `malloc()`, possiamo calcolare il prodotto di due matrici di dimensione determinata nel corso dell'esecuzione:

```
#include<stdlib.h>
#include<stdio.h>
...
int i,j,h;
int I,H,J;
float * A;
float * B;
float * C;
...
scanf("%d",&I); // acquisisce da tastiera il valore di I, J, e H
scanf("%d",&H);
scanf("%d",&J);
...
A=(float *)malloc(I*H*sizeof(float)); // alloca i tre array
B=(float *)malloc(H*J*sizeof(float));
C=(float *)malloc(I*J*sizeof(float));

for(i=0;i<I;i++)
    for(j=0;j<J;j++)
    {
        C[i*J+j]=0;
        for(h=0;h<H;h++)
            C[i*J+j]+=A[i*H+h]*B[h*J+j];
    }
}
```

A, B e C sono dichiarati come puntatori a float. Con le istruzioni `scanf` vengono acquisite da tastiera le dimensioni delle matrici da rappresentare. La chiamata alla funzione `malloc` alloca lo spazio di memoria. Consideriamo in particolare l'istruzione:

```
A=(float *)malloc(I*H*sizeof(float));
```

`malloc()` richiede al sistema operativo un segmento di memoria di tanti bytes quanto è indicato nel suo argomento. In questo caso `I*H*sizeof(float)` bytes: `I*H` è il numero di locazioni che vogliamo allocare; `sizeof(float)` restituisce la dimensione di un float, ovvero la dimensione di ciascuna locazione. La funzione

`malloc` restituisce il puntatore al primo byte del segmento allocato. Lo restituisce nel tipo di puntatore a carattere. Il cast (`float *`) converte il puntatore nella forma di puntatore a `float` che è quello che serve nel nostro caso.

Una volta allocato l'array e assegnato ad `A` il suo puntatore, gli elementi di `A` sono referenziati con l'indice suffisso entro parentesi quadre (ad esempio `A[i*H+h]`), nello stesso modo con cui potremmo farlo se `A` fosse stato dichiarato staticamente. Si osservi che questa non è una regola speciale ma è piuttosto una applicazione diretta della sintassi e semantica del riferimento a variabile array nella forma `<expr1>[<expr2>]`, dove `<expr1>` è una espressione che restituisce un puntatore (nel nostro caso l'espressione è costituita dal solo riferimento `A`) mentre `<expr2>` è un'espressione che restituisce un intero (nel nostro caso l'espressione è `i*H+h`).

È utile soffermarsi a riflettere sulla differenza che corre tra una variabile array allocata staticamente con una dichiarazione del tipo `Xs[128];`, e una variabile puntatore che viene dichiarata con un `float * Xd;` e alla quale viene poi agganciato un segmento di memoria con una assegnazione del tipo `Xd=(float *)malloc(128*sizeof(float));`. Se riguardate come espressioni, sia `Xs` che `Xd` restituiscono un valore di tipo indirizzo di `float`; tuttavia `Xs` è una costante, non modificabile nell'esecuzione del programma, mentre `Xd` è una variabile che può apparire a sinistra di un assegnamento, come del resto avviene nella chiamata a `malloc()`. Il valore di `Xs` punta fino dall'inizio un segmento riservato di 128 `float`; viceversa il valore di `Xd` è inizialmente impredicibile, e ad esso non è riservato alcun segmento di memoria (per inciso, l'uso di tale indirizzo in mancanza di una inizializzazione è uno degli errori che più frequentemente conducono al crash di un programma). Il segmento che sarà allocato con la chiamata a `malloc()` non è contiguo alle altre variabili del programma, ma si trova in una area gestita attraverso il sistema operativo.

La funzione `malloc()` può fallire se il sistema operativo non dispone di un segmento di memoria contigua dell'ampiezza richiesta. In tal caso la funzione `malloc()` restituisce il valore `NULL`. Che è lo 0 dei puntatori ed è definito da una direttiva `#define` dentro la libreria `<memory.h>`. Nella fase iniziale di sviluppo di un programma è corretto ignorare il problema, cosa che noi faremo spesso per il bene della leggerezza del codice negli esempi. Nella realizzazione della forma definitiva di un programma il problema deve però essere trattato introducendo dei controlli che impediscono al programma di fare uso delle variabili allocate nel caso in cui l'allocazione sia fallita. Nell'esempio sopra, se l'allocazione del segmento per l'array `C` fallisse, gli assegnamenti `C[i*J+j]=0` e `C[i*J+j]+=A[i*H+h]*B[h*J+j]` colpirebbero una locazione di memoria non riservata all'uso, producendo il fal-

limento del programma. Per evitare il problema, l'uso delle variabili allocate dinamicamente deve essere protetto con una guardia:

```
#include<memory.h>
...
float * C;
...
C=(float *)malloc(I*J*sizeof(float));
if(C==NULL)
{ printf(...) // l'allocazione \e fallita
}
else
{ ...           // sezione di codice che usa C
}
```

Un segmento di memoria allocato con la funzione `malloc()` resta in uso fino al termine del programma. Nel caso in cui il segmento cessi di essere utile prima del termine del programma, può essere rilasciato attraverso l'uso della funzione `free()`.

```
#include<memory.h>
...
float * C;
...
C=(float *)malloc(I*J*sizeof(float));
...
free(C);
```

Il rilascio anticipato della memoria allocata con `malloc()` è un dettaglio che spesso viene lasciato alle fasi più avanzate dello sviluppo. Può diventare rilevante nel caso di programmi che allocano frequentemente strutture temporanee. In alcuni linguaggi, e come caso notevole in Java, la memoria allocata ma non più raggiungibile attraverso alcun puntatore in uso nel programma viene automaticamente recuperata attraverso un meccanismo di garbage collection che rimane trasparente al programmatore.

Arrays e ripetizioni

In chiusura, è utile soffermarsi a riflettere sul ruolo e la rilevanza generale degli array ai fini della capacità espressiva di un linguaggio di programmazione.

Consideriamo per esempio il caso del calcolo del prodotto di matrici. Con le stesse istruzioni è possibile calcolare il prodotto di matrici di dimensione diversa. Il che significa che programmi con lo stesso numero di istruzioni possono trattare dati di dimensionalità diversa.

Se osserviamo la struttura del codice vediamo che questo è reso possibile dall'uso congiunto di arrays e istruzioni di iterazione (il `for`, nell'esempio): la stessa istruzione all'interno di un loop tratta variabili diverse in momenti diversi dell'esecuzione. Questo è reso possibile dal fatto che l'array ammette un campo variabile, l'indice, nella costruzione del riferimento a una variabile. Se questo non fosse possibile, ciascuna variabile dovrebbe essere referenziata con un suo nome individuale, costante, e il programma dovrebbe includere un numero di istruzioni almeno pari al numero di variabili trattate. In una prospettiva teorica, un tale programma non è nemmeno riguardabile come un algoritmo. Nella pratica, esso risulta assolutamente inadeguato a trattare alcun caso significativo.

Questa è la ragione sostanziale per cui gli array di variabili esistono in qualsiasi linguaggio di programmazione, al di là di dettagli sul modo con cui sono memorizzati o indicizzati. Ed è la ragione per cui il trattamento di arrays trova un supporto fino dal livello dell'hardware della CPU e del linguaggio macchina che questa riconosce.

1.5.5 Istruzioni

Le istruzioni servono a dirigere il flusso di esecuzione di un programma. Controllano l'ordine con cui sono eseguite le espressioni e quindi l'ordine con cui queste producono side effects sui dati codificati nelle variabili.

Espressione

L'istruzione elementare è un'espressione seguita da un punto e virgola:

<statement> ::= <expr>; | ...

La sua semantica consiste nel calcolare l'espressione per poi passare il controllo all'istruzione successiva.

Sequenze

Più istruzioni possono essere combinate in sequenza e/o raccolte entro un compound:

*<statement> ::= ...
| <statement₁><statement₂>
| {<statement₁>}
| ...*

La semantica della sequenza *<statement₁><statement₂>* consiste nell'eseguire *<statement₁>*, poi *<statement₂>*, e infine passare il controllo all'istruzione successiva. Si noti che la ripetizione della composizione in sequenza permette di avere una istruzione composta di un qualsiasi numero di istruzioni.

La semantica del compound {*<statement₁>*} consiste nell'eseguire l'istruzione interna al compound e poi passare il controllo alla prima istruzione successiva alla parentesi di chiusura.

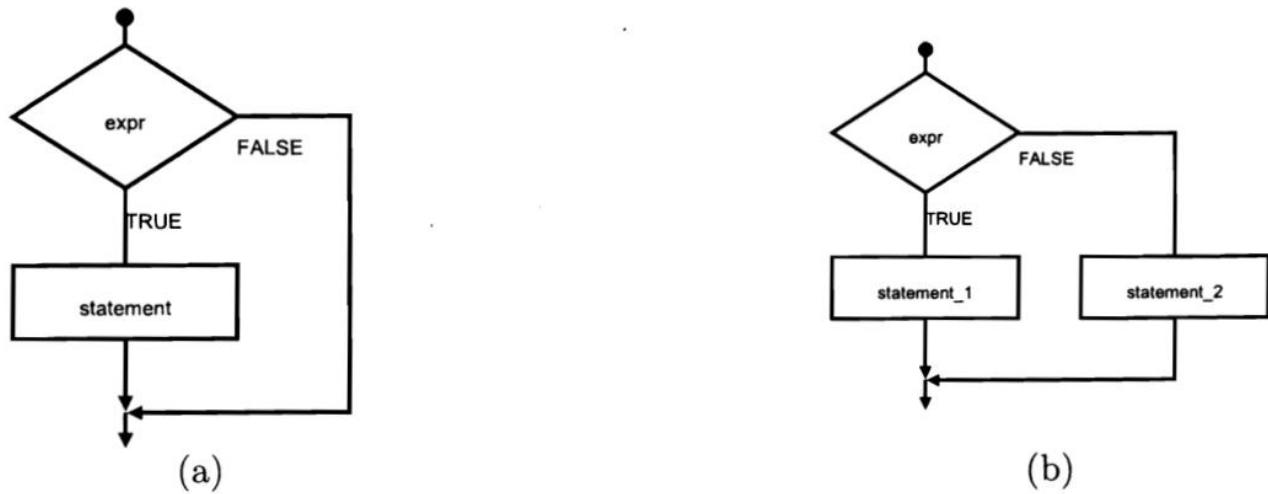


Figura 1.18: Controllo del flusso nelle istruzioni a) if semplice e b) if-else.

Condizione

L'istruzione `if` permette di condizionare l'esecuzione di una istruzione, detta corpo, al valore restituito da un'espressione, detta guardia:

```

<statement> ::= ...
| if(<expr>)<statement1>
| if(<expr>)<statement1>else<statement2>
| ...
  
```

Fig.1.18a illustra la semantica dell'istruzione `if`. Inizialmente, viene calcolata l'espressione `<expr>`. Se il valore restituito è diverso da zero (ovvero se è `true`) viene eseguita l'istruzione `<statement1>`. In entrambi i casi il controllo è passato all'istruzione successiva. Nell'istruzione `if-else`, illustrata in Fig.1.18b, se la guardia è falsa viene eseguito un corpo alternativo `<statement2>`.

Si noti che le istruzioni `<statement1>` e `<statement2>` possono in realtà essere a loro volta istruzioni composte. In particolare, l'uso del compound permette che `<statement1>` e `<statement2>` siano composte da più istruzioni in sequenza.

Iterazione

Le istruzioni di iterazione (loop) permettono di eseguire ripetivamente un corpo di istruzioni fino al momento in cui una espressione di guardia diventa falsa. Esistono tre diverse istruzioni di iterazione, `for`, `while` e `do-while`, che differiscono nel modo in cui è espressa la guardia:

```

<statement> ::= ...
| for(<expr_init>;<expr_guard>;<expr_inc>)<statement>
| while(<expr_guard>)<statement>
| do<statement>while(<expr_guard>);
| ...

```

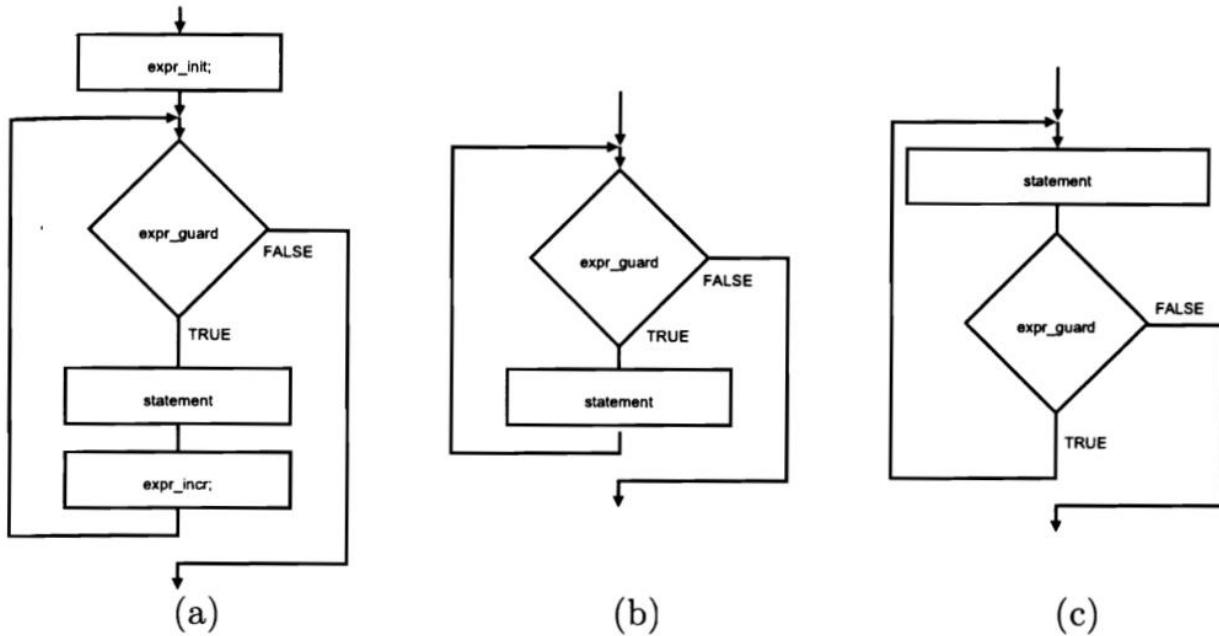


Figura 1.19: Controllo del flusso nelle istruzioni di iterazione: a) for; b) while; c) do-while.

La semantica dell'istruzione **for** è illustrata in Fig.1.19a. Inizialmente viene eseguita l'espressione di inizializzazione $\langle \text{expr}_{\text{init}} \rangle$. Viene poi calcolata l'espressione di guardia $\langle \text{expr}_{\text{guard}} \rangle$. Se il risultato restituito è diverso da 0 (ovvero se è true), viene eseguito il corpo $\langle \text{statement}_1 \rangle$, viene calcolata l'espressione di incremento $\langle \text{expr}_{\text{inc}} \rangle$, e il controllo viene riportato all'espressione di guardia. se invece il risultato restituito è uguale a 0, (ovvero se è false), allora l'iterazione termina e il controllo è passato alla prima istruzione successiva.

Si osservi che la guardia è controllata prima del corpo; come caso particolare se al primo controllo l'espressione di condizione risulta falsa, l'istruzione termina senza avere mai eseguito il corpo.

Nell'istruzione **while**, illustrata in Fig.1.19b, viene inizialmente calcolata l'espressione di guardia; se il valore restituito è diverso da 0 (ovvero se è true), viene eseguito il corpo e il controllo viene riportato all'espressione di guardia; quando l'espressione di guardia restituisce un valore false l'iterazione termina e il controllo è passato all'istruzione successiva.

Si osservi che anche in questo caso la guardia viene controllata in testa ed è possibile che il corpo non venga eseguito nemmeno una volta.

Nell'istruzione `do-while`, illustrata in Fig.1.19c, viene eseguito il corpo e poi è calcolata la guardia; se la guardia restituisce il valore vero il controllo torna all'esecuzione del corpo, se invece è falso il controllo passa all'istruzione successiva.

Si osservi che, diversamente da `for` e da `while`, con `do-while`, la guardia è controllata in coda e il corpo è sempre eseguito almeno una volta.

I tre costrutti di iterazione `for`, `while`, e `do-while` sono dipendenti, nel senso che con uno è possibile sostituire gli altri. Tuttavia ciascuno di essi esprime un diverso tipo di controllo. L'uso del costrutto più appropriato rende comprensibile la logica del controllo del programma.

L'istruzione `for` è adatta a strutture di controllo che risultano in un numero di iterazioni determinato e noto fino dal momento della prima iterazione. Viceversa `while` e `do-while` permettono di creare strutture di controllo in cui il numero di iterazioni non è predicibile a priori.

Mentre `while` effettua il controllo in testa, `do-while` lo effettua in coda. Per questo con `while` è possibile che il corpo dell'iterazione non sia mai eseguito mentre con `do-while` il corpo viene sempre eseguito almeno una volta. Con `while` il controllo è basato su una condizione che deve essere determinata prima di raggiungere l'istruzione, mentre con `do-while` la condizione può essere determinata dalle istruzioni del corpo dell'iterazione.

Sia `while` che `do-while` garantiscono in uscita che la condizione di guardia è falsa. In aggiunta `while` garantisce anche che la condizione di guardia è vera prima dell'esecuzione del corpo dell'iterazione.

Salto

Le istruzioni di salto permettono di trasferire il controllo a istruzioni non continue. Sono istruzioni abbastanza macchinose da descrivere formalmente, sia nella sintassi che nella semantica. E sono anche istruzioni di poca eleganza, spesso da evitare in una programmazione strutturata a modo. Alcune di esse, `switch` e `break`, sono però impegnate largamente nella pratica della programmazione c:

```
<statement> ::= ...  
          | goto <label>;  
          | switch(<expr>)  
            { {case <const>: <statement>}  
             [default: <statement>]  
            }  
          | break;  
          | continue;  
          | ...
```

dove *<label>* ha lo stesso formato del nome di una variabile. Si noti che nell'espressione della sintassi di `switch` le parentesi { } e [] denotano i simboli di ripetizione e di opzionalità del metalinguaggio BNF, da non confondere con le parentesi { } e [] del linguaggio c.

L'istruzione `goto label;` passa il controllo alla prima istruzione successiva alla riga di programma in cui appare l'etichetta *<label>*. Un esempio concreto del `goto`, e' esemplificato nel seguito assieme alle istruzioni `break` e `continue`.

L'istruzione `goto` è sostanzialmente equivalente al `jump` incondizionato di un qualsiasi linguaggio assembler. In congiunzione con l'istruzione condizionale `if`, equivalente al `conditional branch` dell'assembler, permetterebbe la programmazione di qualsiasi logica di controllo. Tuttavia, l'uso del `goto` in un linguaggio di alto livello può essere evitato ed è in genere assolutamente sconsigliato. Non è solo un fatto di stile: l'uso del `goto` complica la comprensione della logica di esecuzione di un programma e impedisce di reagire sulla sua correttezza. Per almeno due ragioni. Da un lato l'uso del `goto` lascia indeterminata l'istruzione che è stata eseguita prima di arrivare ad una label. In linea di principio, questa può essere l'istruzione che precede l'etichetta, ma può anche essere un `goto` posizionato in qualunque riga della sezione di codice da cui è possibile raggiungere l'etichetta. Dall'altro lato, l'uso del `goto` permette di entrare e uscire dai limiti del corpo di una istruzione condizionale o di iterazione senza passare per il controllo della guardia. E questo impedisce di fare affidamento sulle condizioni che la guardia assicura per l'ingresso e l'uscita dal corpo di una istruzione condizionale o di iterazione.

L'istruzione `switch` è un `goto` aritmetico. C'è una guardia in testa (*<expr>*) che deve restituire un valore intero, e un corpo delimitato da parentesi graffe, che contiene un numero di etichette, ciascuna associata a un diverso valore costante (*<const>*) di tipo intero. È opzionale l'etichetta *<default>*. Nell'esecuzione viene inizialmente calcolata l'espressione *<expr>*. Se il valore restituito è uguale a quello di una delle costanti che appaiono in una etichetta `case`, il controllo viene trasferito a quella etichetta e l'esecuzione prosegue fino al termine del corpo dell'istruzione. Se nessuna delle costanti coincide con il valore restituito allora il controllo è trasferito alla chiusura del corpo dello `switch`, oppure all'etichetta `default`: quando questa è espressa nell'istruzione.

Un esempio aiuta la comprensione:

```
switch(a+5)
{ case 2:
    b=10;
  case 4:
    c=20;
```

```
    default:  
        d=30;  
}
```

Il calcolo dell'espressione di guardia `a+5` restituisce un valore. Se il valore è 2, allora il controllo è trasferito all'etichetta `case 2:` e l'esecuzione prosegue con le istruzioni `b=10; c=20; d=30;;`. Se invece il valore è 4, allora sono eseguite le istruzioni `c=20; d=30;;`. Infine, se il valore restituito non è né 2, né 4, allora viene eseguita la sola istruzione `d=30;;`. In tutti i tre casi, il controllo viene poi trasferito all'istruzione successiva alla chiusura del corpo dello `switch`.

Rispetto a `goto`, l'istruzione `switch` ha il pregio di localizzare entro il limite di un `compound` tutte le labels a cui è possibile saltare. Questo limita il problema di indeterminazione della istruzione precedente e di violazione dei limiti del corpo di un condizionale o di una iterazione. L'istruzione `switch` è molto usata nella programmazione c, quasi sempre in congiunzione con l'istruzione `break`. L'uso combinato delle due istruzioni permette di realizzare in modo chiaro frammenti di codice nei quali una di più procedure (transactions) alternative viene selezionata in base al valore di una variabile.

L'istruzione `break` può apparire solo all'interno del corpo di una istruzione di iterazione o di uno `switch`. L'effetto di `break` è quello di trasferire il controllo alla prima istruzione successiva all'istruzione di iterazione o di `switch` nella quale è contenuta.

L'istruzione `break` è ampiamente usata nella pratica della programmazione c. In particolare, risulta quasi essenziale per l'uso dell'istruzione `switch`: ciascuna delle clausole associate alle diverse etichette `case` viene chiusa con un `break`. In tal modo, le istruzioni associate alle diverse etichette sono eseguite in maniera alternativa, piuttosto che incrementale. Illustriamo il concetto modificando l'esempio precedente:

```
switch(a+5)  
{ case 2:  
    b=10;  
    break;  
 case 4:  
    b=20;  
    break;  
 default:  
    b=30;  
}
```

questa volta, se l'espressione di guardia restituisce il valore 2, viene eseguita l'istruzione `b=10;` e poi il controllo è trasferito all'istruzione successiva allo `switch`.

Se invece il valore restituito è 4, viene eseguita l'istruzione `b=20`; se il valore non è né 2 né 4, viene eseguita l'istruzione `b=30`.

Il costrutto `break` è funzionalmente equivalente a un `goto` diretto ad una label immediatamente successiva alla chiusura del corpo dello `switch`. Riportiamo il codice con l'obiettivo di fornire un esempio concreto di uso del `goto`, peraltro già ampiamente sconsigliato nella discussione precedente:

```
switch(a+5)
{ case 2:
    b=10;
    goto after_switch_body;
  case 4:
    b=20;
    goto after_switch_body;
  default:
    b=30;
}
after_switch_body:
```

L'istruzione `continue` può apparire solo all'interno del corpo di una istruzione di iterazione. Il suo effetto è quello di saltare al termine della iterazione corrente. In genere è possibile evitarne l'uso a vantaggio di costrutti di maggiore chiarezza e verificabilità.

Esempi

Come esempio elementare consideriamo il calcolo della somma dei valori memorizzati su un'array: un sommatore è inizializzato a 0 e poi incrementato con i valori di ciascun elemento dell'array. Il numero di elementi dell'array, e quindi il numero di iterazioni da eseguire, sono noti dalla prima iterazione. Anche in questo caso la struttura di controllo può essere creata convenientemente usando un `for`:

```
main()
{
  float A[100];
  int count;
  float sum;
  ...
  sum=0;
  for(count=0;count<100;count=count+1)
    sum=sum+A[count];
}
```

L'esempio illustra una caratteristica tipica nella prassi di uso del C: nell'iterazione `for`, il contatore parte tipicamente da 0 (l'espressione di inizializzazione è `count=0`)

ed è limitato da un minore secco rispetto al numero dei dati da trattare e quindi delle iterazioni da compiere (l'espressione di controllo è `'count<100'`).

Le tre espressioni di guardia di un'istruzione di iterazione `for` sono generiche espressioni. Il linguaggio C permette di farne l'uso più appropriato al caso. L'espressione di inizializzazione può essere estesa a raccogliere più condizioni iniziali. L'espressione di incremento può essere aggiustata per contare all'indietro, o per selezionare solo alcuni valori lungo una qualsiasi successione. Il seguente frammento di codice calcola la somma degli elementi dell'array A che si trovano nelle posizioni che sono potenze del 2. Si noti che l'inizializzazione a 0 della variabile `sum` su cui si calcola la somma è portata dentro alla clausola di inizializzazione del `for`; si noti anche che l'incremento dell'variabile `sum` è riscritto in forma compatta `sum+=A[count]`:

```
float A[100];
int count;
float sum;
...
for(sum=0,count=1;count<100;count=count*2)
    sum+=A[count];
```

La flessibilità nell'uso delle tre espressioni permette di produrre dei mostri, di limitata applicazione pratica. Ad esempio è possibile generare un loop-forever se la espressione di controllo è fatta in maniera tale da restituire sempre 1.

Questo riflette una caratteristica generale dell'architettura del linguaggio C: il linguaggio definisce limiti ampi entro i quali ricadono sia i patterns di uso comune che una infinità di altre cose che non vale la pena di enumerare; molte sono inutili, altre potenzialmente pericolose per la correttezza del programma. Alcune possono risultare risolutive in particolari condizioni. L'effetto di una sintassi lasca di questo tipo è quello di limitare la complessità delle regole del linguaggio piuttosto che la complessità dei programmi che il linguaggio può generare. Questo diventa un vantaggio se l'apprendimento del linguaggio è finalizzato alla comprensione delle regole essenziali, piuttosto che all'enumerazione dei diversi casi che derivano dalla loro declinazione.

Come esempio ulteriore consideriamo il problema di stampare gli elementi di un array fino ad identificarne uno di segno positivo, che non deve essere stampato. In questo caso non è a priori determinato il numero di elementi che dovranno essere stampati. È dunque più appropriato l'uso di un `while` o un `do-while`. Potrebbe essere il caso che non si debba stampare nemmeno un elemento, e quindi è conveniente usare un `while`:

```
int position;
```

```

float A[100];

position=0;
while(A[position]<=0)
{ printf("\%d", A[position]);
  position++;
}

```

L'esempio illustra bene l'essenza del costrutto `while`: è una trappola che non rilascia il controllo fino al momento in cui diventa falsa la condizione di guardia. È un ottimo costrutto per garantire quella che formalmente viene chiamata correttezza parziale: se il programma termina, allora il risultato che produce è corretto. Per ottenere quella che si chiama correttezza totale occorre garantire anche che l'iterazione termini.

L'esempio sopra non garantisce la terminazione: se tutti gli elementi dell'array sono negativi, il programma non esce mai dal loop, e arriverà a stampare valori più o meno casuali ottenuti quando `position` eccede il valore massimo 99. Per correggere il problema occorre mettere nella guardia un controllo aggiuntivo sulla possibile esaustione dell'array:

```

int position;
float A[100];

position=0;
while(A[position]<=0 && position<100)
{ printf("\%d",A[position]);
  position++;
}

```

Così facendo, è garantito che l'iterazione termina entro un massimo di 100 iterazioni, anche se la condizione garantita in uscita è un pò più debole: è garantito che sia falso `A[position]<=0 && position<100`; per le leggi di De Morgan, questo implica che è vera una almeno delle due condizioni `A[position]>0` e `position==100`.

La struttura di controllo dell'esempio sopra ha qualcosa di generale. Usando un termine dell'ingegneria del software diremmo che esso costituisce un pattern che può applicarsi a una varietà di casi. Le due condizioni testate nella guardia sono una condizione di legalità e una di terminazione. La condizione `position<100` garantisce la legalità della variabile `A[position]`, e più in generale delle istruzioni che stanno nel corpo del `while`. La condizione `A[position]<=0` controlla la terminazione per successo della ricerca. All'interno del corpo ci sono due istruzioni, una di manipolazione del dato che è il vero contenuto dell'iterazione,

l'altra di incremento del contatore che è in qualche modo parte del controllo della sequenzializzazione.

Come ultimo esempio consideriamo il caso di una trappola che acquisisce numeri da tastiera fino ad ottenerne uno di segno non negativo. Anche in questo caso il numero di iterazioni non è noto al momento della prima iterazione, per cui non è conveniente l'uso di un `for`. Rispetto all'esempio precedente, in questo caso un `do-while` risulta più appropriato perché la condizione di terminazione si determina all'interno dell'istruzione di lettura da tastiera che sta nel corpo dell'iterazione:

```
int number;  
  
do  
{   scanf("\%d", &number); // legge number da tastiera  
}while(number<0);
```

Si osservi il fatto che `do-while` ha il controllo in coda impedisce di avere un controllo di legalità a monte delle istruzioni che stanno nel corpo dell'iterazione. questo spesso richiede l'aggiunta di un controllo preliminare realizzato con un `if`.

1.5.6 Funzioni

In prima approssimazione la funzione è un costrutto di controllo del flusso: è una sorta di `goto` con ritorno che permette di trasferire il controllo a una sezione di codice e riceverlo indietro quando la sezione è terminata. La stessa sezione può essere chiamata da punti diversi ed è comunque capace di restituire il controllo al particolare punto dal quale è stata chiamata. Sono possibili, e anche consuete, chiamate nidificate e chiamate ricorsive per le quali una sezione viene richiamata in maniera *rientrante* prima della sua terminazione (vedi Fig.1.20).

La reale potenza delle funzioni non si limita però alla possibilità di partizionare il programma in segmenti, ma consiste piuttosto nella combinazione di questo con la capacità di separare e disciplinare la condivisione delle variabili referenziate nelle diverse sezioni. Questo fornisce un meccanismo di *information hiding* che permette di sviluppare una funzione senza il bisogno di conoscere completamente la logica interna e le variabili su cui operano le altre funzioni del programma.

Di per sé la funzione non aumenta la potenza espressiva del linguaggio, ma fornisce uno strumento di strutturazione e partizionamento del codice che gioca un ruolo essenziale nello sviluppo (e nella manutenzione) di qualunque applicazione realistica.

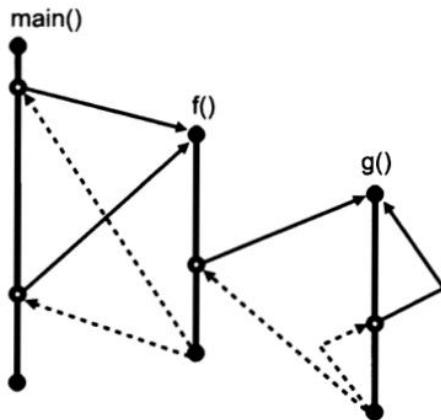


Figura 1.20: Il trasferimento di controllo tra le sezioni di tre funzioni `main()`, `f()` e `g()`. La funzione `main()` invoca `f()` da due diversi punti del suo corpo; alla chiamata (linee fini piene), il controllo viene trasferito da `main()` a `f()` e poi viene restituito (linee punteggiate) allo specifico punto da cui era avvenuta l'invocazione. La funzione `f()` a sua volta invoca `g()`, che contiene una invocazione ricorsiva a sé stessa.

Parametri formali, attuali e loro legame

Un programma c è costituito da un insieme di funzioni, ciascuna delle quali è una sezione di codice associata a un nome e costituita da una sequenza di istruzioni.

Nell'esecuzione del programma il controllo è inizialmente attribuito a una funzione principale che ha nome `main()`, ed è poi trasferito ripetutamente da una funzione all'altra attraverso chiamate e ritorni del controllo. La chiamata è effettuata facendo riferimento al nome della funzione che deve ricevere il controllo, mentre il ritorno avviene quando nella funzione chiamata viene eseguita una istruzione `return` oppure sono esaurite le istruzioni da eseguire. L'intero programma termina quando la funzione principale termina e restituisce il controllo.

Ciascuna funzione manipola variabili di due tipi: le *variabili locali* dichiarate all'interno della funzione stessa, e i *parametri formali* che sono determinati al momento in cui la funzione riceve il controllo (a dire il vero esistono anche *variabili globali*, di cui parliamo, male, nel seguito).

Al momento di eseguire una chiamata, una funzione specifica quali tra le sue variabili devono essere *legate* come *parametri attuali* ai parametri formali sui quali opera la funzione chiamata. Attraverso questo legame, i riferimenti ai parametri formali che compaiono all'interno della funzione chiamata sono risolti in maniera da referenziare in maniera più o meno diretta i parametri attuali. Il legame di parametri attuali e formali serve a realizzare una comunicazione bidirezionale tra funzione chiamante e chiamata nella quale sono scambiati valori di ingresso sui

quali applicare la computazione della funzione chiamata e risultati prodotti nella computazione stessa. La risoluzione può avvenire secondo due diverse *tecniche di legame* dei parametri: *per riferimento* e *per valore*.

Nel legame per riferimento, ogni riferimento della funzione chiamata ad un parametro formale colpisce direttamente il parametro attuale che gli è stato legato nella chiamata. Così facendo la funzione chiamata condivide con la funzione chiamante le variabili che questa ha specificato come parametri attuali. Il legame per riferimento è supportato in c++ e Java, ma non nel c.

Nel linguaggio c, i parametri sono sempre legati per valore. In questa tecnica, al momento della chiamata viene creata una variabile temporanea su cui è copiato il valore del parametro attuale al momento della chiamata, e i riferimenti ai parametri formali nella funzione chiamata colpiscono le variabili temporanee in cui sono stati copiati i parametri attuali rispettivi.

Nel legame per valore, la funzione chiamata non ha accesso alle variabili della funzione chiamante ma solo a una loro copia. Mentre questo permette al chiamante di passare valori di ingresso al chiamato, non permette invece al chiamato di accedere direttamente alle variabili del chiamante per lasciare traccia dei risultati prodotti nella computazione effettuata. La limitazione è superata attraverso l'uso degli indirizzi: per permettere a una funzione di modificare una variabile le viene passata una copia dell'indirizzo della variabile stessa; conoscendo l'indirizzo della variabile da modificare, la funzione chiamata può colpirla attraverso l'operatore di de-referenziazione *. La soluzione, solo apparentemente macchinosa, permette un controllo diretto sulla separazione delle variabili tra funzioni chiamante e chiamata: il chiamante ha sempre la garanzia che il chiamato non modifica le variabili passate come parametri attuali, ma solo le variabili che sono raggiungibili a partire dagli indirizzi specificati tra i parametri attuali stessi.

Stack di sistema, tempo di vita e visibilità delle variabili

In generale, il termine stack (pila o catasta, in inglese) denota una struttura di memoria nella quale sono dinamicamente memorizzati e estratti valori con una politica *Last In First Out*: l'ultimo valore memorizzato è il primo ad essere estratto. Questo si realizza attraverso un vettore di valori e un indice che denota la posizione dell'ultimo valore inserito. L'operazione di inserimento, denominata *push*, è realizzata aumentando il valore del TOS e inserendo il nuovo valore nella posizione puntata dal TOS stesso. L'estrazione, denominata *pop*, è eseguita estraendo il valore puntato dal TOS e poi riducendo il valore del TOS stesso (vedi Fig.1.21). Lo stack di sistema è un'area di memoria che supporta i meccanismi di trasferimento del controllo e la comunicazione tra le funzioni. Tale area di memoria rimane trasparente al programmatore, ma la comprensione del suo funzionamento

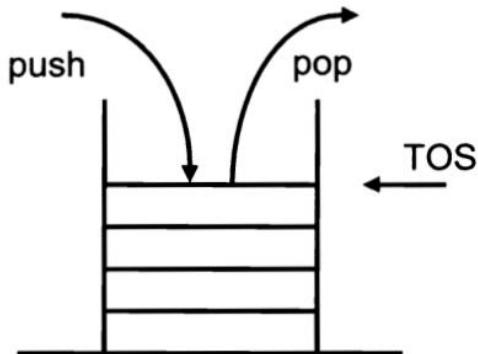


Figura 1.21: Lo stack e' una struttura di memorizzazione First In Last Out.

permette di comprendere in maniera motivata le regole circa la visibilità e il tempo di vita delle variabili.

Al momento della chiamata, sullo stack viene memorizzato l'indirizzo dell'ultima istruzione eseguita nella funzione chiamante, che serve al termine della funzione chiamata per determinare l'indirizzo dell'istruzione a cui deve essere restituito il controllo. Sullo stack viene poi riservato lo spazio per i parametri formali della funzione chiamata e tali parametri sono inizializzati con una copia dei parametri attuali a loro collegati. Infine, sempre sullo stack, sono allocate le variabili dichiarate all'interno della funzione chiamata.

Poiché i parametri sono allocati sullo stack al momento della invocazione di una funzione, nel caso di una invocazione ricorsiva, allo stesso segmento di codice sono associati diversi spazi di indirizzamento dei dati collocati in diversi segmenti dello stack. Questo permette di avere contemporaneamente attivate più istanze di una funzione che eseguono le stesse righe di codice operando su dati diversi.

Il *tempo di vita* (lifetime) di una variabile è l'intervallo temporale durante il quale una variabile è mantenuta in memoria. Per come funziona lo stack di sistema, il tempo di vita delle variabili locali e dei parametri attuali di una funzione inizia al momento in cui la funzione è attivata e termina al momento in cui la funzione restituisce il controllo. Dunque in qualunque istante sono in vita tutte e sole le variabili locali e i parametri attuali della funzione che ha il controllo e di tutte le funzioni che la precedono nella sequenza di chiamata.

La *visibilità* (scope) di una variabile è la sezione di codice entro la quale è possibile fare riferimento alla variabile stessa. In C, una variabile può essere referenziata solo nel corpo della funzione di cui essa è un parametro formale o una variabile locale.

È utile osservare che l'assunzione che una funzione non possa fare riferimento alle variabili nello spazio degli indirizzi di una funzione da essa chiamata riflette

un limite fisico: nel momento in cui una funzione è attiva, non sono invece attive le funzioni da essa chiamate.

Viceversa, l'assunzione che la funzione chiamata non possa fare riferimento alle variabili nello spazio degli indirizzi della sua funzione chiamante è un vincolo aggiunto dal linguaggio che in principio potrebbe essere rilassato: in effetti quando una funzione è attivata sono in vita sullo stack anche le variabili della sua funzione chiamante che quindi potrebbero essere rese visibili se il compilatore volesse permetterlo (questa è ad esempio la disciplina del Pascal). L'assunzione di una totale separazione degli spazi degli indirizzi è un limite intenzionale posto dal linguaggio con l'obiettivo di ridurre l'accoppiamento tra funzioni.

La direttiva `static` permette di rendere staticamente allocata una variabile, anche se questa è dichiarata all'interno di una funzione. Una variabile statica è visibile solo all'interno della sua funzione ma resta in vita dal momento della prima attivazione della funzione fino al termine del programma. È inizializzata una sola volta ed è condivisa tra istanze diverse nella chiamata alla funzione. Concretamente può servire a distinguere la prima chiamata della funzione rispetto alle successive, o per contare il numero di volte che una funzione è attivata nel corso della computazione.

Variabili globali e valori restituiti

La comunicazione tra funzioni chiamante e chiamata oltre che sul legame dei parametri può fare leva su due ulteriori meccanismi del linguaggio c: le variabili globali e il valore di ritorno.

Una *variabile globale* è una variabile dichiarata all'esterno di qualsiasi funzione. Una tale variabile non viene allocata sullo stack di sistema bensì in un'area di memoria nel segmento dei dati del programma. Ha tempo di vita coincidente con l'intera esecuzione del programma ed è visibile in qualsiasi funzione successiva al punto di dichiarazione.

Le variabili globali semplificano la comunicazione tra funzioni evitando la complessità del passaggio dei parametri. Anche aumentano la efficienza della programmazione evitando la copia dei valori attraverso lo stack. Tuttavia le variabili globali costituiscono un fattore di degradazione della comprensibilità e manutenibilità del codice. Sono da impiegarsi solo in condizioni limite e devono essere accompagnate da un livello di documentazione elevato.

Una funzione può restituire un valore (di un tipo). Tale *valore restituito* è un meccanismo di comunicazione unidirezionale dal chiamato al chiamante. È molto usato nella pratica della programmazione, e non ha particolari controindicazioni

salvo il fatto che non permette di comunicare più di un valore. Noi lo usiamo poco. Per ridurre la molteplicità delle soluzioni ai problemi che affrontiamo. Ma anche perché spesso il valore di ritorno è convenientemente riservato a riportare l'esito nell'esecuzione di una funzione al fine della gestione delle eccezioni (error handling).

Dichiarazione

La dichiarazione di funzione, usualmente nota come *prototipo*, istruisce il compilatore circa il nome simbolico associato alla funzione, il numero e il tipo dei parametri, il tipo del valore restituito. Questo serve al compilatore per generare il codice che copia i parametri attuali sullo stack e riceve il valore di ritorno della funzione.

La sintassi della dichiarazione di funzione è parte della categoria delle dichiarazioni già introdotta nel capitolo 1.5.1:

```
<declaration> ::= <type><decl>;
```

ed è ottenuta estendendo la categoria sintattica *<decl>*:

```
<decl> ::= ... | <decl>(<formal_parameter_list>)
```

dove *<formal_parameter_list>* è il tipo void oppure è una sequenza di dichiarazioni di variabili separate da virgolette:

```
<formal_parameter_list> ::= void | <type><decl>{ , <type><decl> }
```

Si noti che le parentesi { } denotano la ripetizione del metalinguaggio BNF, e non le parentesi { } del linguaggio c.

Per quanto attiene alla semantica, il significato della dichiarazione

```
<declaration> ::= <type><decl>(<formal_parameter_list>);
```

è specificato riguardando la parentesi tonda come un *modificatore suffisso* della dichiarazione

```
<declaration_base> ::= <type><decl>;
```

Specificamente, declaration dichiara una funzione che ha lo stesso nome della variabile che sarebbe dichiarata in declaration_base, prende in ingresso i parametri dichiarati nella formal_parameter_list, e restituisce un valore del tipo della variabile che sarebbe dichiarata in declaration_base.

In modo del tutto simile a quanto avviene per gli arrays, il nome associato alla funzione denota l'indirizzo a partire dal quale la funzione è memorizzata.

È utile osservare che l'effetto del modificatore suffisso () è contemplato nella regola pratica che abbiamo introdotto nel capitolo 1.5.4: a partire dal nome dell'oggetto dichiarato si procede verso destra fino a che esistono modificatori e poi si procede verso sinistra fino a raggiungere il termine type, rispettando eventuali parentesi tonde. In questo schema, il modificatore suffisso (<formal_parameter_list>) viene interpretato con la dizione ...*funzione che riceve in ingresso i parametri specificati in <formal_parameter_list> e restituisce*....

Per esempio:

```
float X(void);           // X e' una funzione che non prende parametri e
                         //   restituisce un float
void Y(int x,float * y); // Y e' una funzione che riceve un int e un indirizzo
                         //   di float e non restituisce valore
float Z[128](void);     // Z e' un array di 128 funzioni che ricevono un
                         //   int e restituiscono un float
float (* W)(void);      // W e' un puntatore a una funzione che non riceve
                         //   parametri e restituisce un float
float (* V[16])(void);  // V e' un array di 16 puntatori a funzione che
                         //   non riceve parametri e restituisce un float
```

Definizione

Mentre la dichiarazione istruisce il compilatore per quanto è necessario a tradurre il codice della sezione chiamante, la *definizione* specifica il codice che costituisce la sezione chiamata. Tale codice è composto dalla dichiarazione delle variabili locali allocate nella funzione chiamata e dalle istruzioni che costituiscono il corpo della funzione chiamata stessa:

```
<function_definition> ::=  <type><decl>(<formal_parameter_list>
{
    {declaration}
    <statement>
}
```

Come già sopra, si osservi che le parentesi {} denotano la ripetizione del meta-linguaggio BNF, da non confondere con le parentesi {} del linguaggio c. Si osservi anche che la riga <type><decl>(<parameter_list>) nella definizione coincide con la dichiarazione della funzione salvo omettere il punto e virgola ';' terminale. Ad esempio:

```
// prototipo:
void swap(float * V,int n1,int n2); //type identifier(<formal_parameter_list>);

// definizione:
```

```

void swap(float * V,int n1,int n2) //type identifier(<formal_parameter_list>
{
    float tmp; // {<declaration>}
    tmp=V[n1]; // <statement>
    V[n1]=V[n2]; //
    V[n2]=tmp; //
}

```

Struttura di un programma

Un programma è strutturato come una sequenza di dichiarazioni e definizioni di funzione

`<program> ::= {<function_definition>|<function_declaration>|...}`

Dichiarazioni e definizioni possono essere comunque intercalate, ma una funzione non può essere definita né referenziata prima della sua dichiarazione. Usualmente le dichiarazioni sono poste in testa al programma. La prima funzione definita è associata al nome `main`.

Il programma è eseguito attribuendo inizialmente il controllo alla funzione `main()` e terminando nel momento in cui questa termina.

Non essendo chiamata da una altra funzione, la funzione `main` usualmente non ha parametri formali né restituisce valori. Per questo il suo prototipo è usualmente `void main(void)`. È utile comunque sapere che è possibile associare la funzione `main` sia a parametri formali che a un valore restituito, che vengono usati per comunicare dati tra il programma e il sistema operativo che lo manda in esecuzione.

Nel programma possono comparire anche direttive, usualmente poste in testa e caratterizzate da un simbolo di testa `#`, istruzioni `typedef`, e dichiarazioni di variabili globali:

```

<program> ::= {
    <function_definition>|
    <function_declaration>|
    <directive>|
    <typedef>|<type> <identifier>;|
    <declaration>
}

```

Esistono varie categorie di direttiva tra cui menzioniamo solo `#include` e `#define`:

`<directive> ::= #include<filename.h> | #define <identifier> <any> | ...`

`#include<filename.h>` include nel file processato dal compilatore le dichiarazioni contenute nel file `filename.h`. Questo permette al programmatore di omettere la dichiarazione di tutte funzioni già dichiarate in `filename.h`.

`#define <identifier> <text_string>` associa il nome simbolico `<identifier>` ad una qualsiasi stringa di testo. Spesso la stringa è una costante, ma potrebbe essere anche il nome di una variabile o una sequenza di istruzioni. Prima dell'avvio del compilatore, un pre-processore sostituisce le istanze del simbolo `<identifier>` con la stringa `<text_string>`.

`typedef <type> <identifier>;` definisce `<identifier>` come alias del tipo `type`. Un uso frequente è la definizione di Boolean come alias di un tipo intero con TRUE e FALSE #definiti uguali a 1 e 0 rispettivamente:

```
typedef unsigned short int Boolean;
#define TRUE 1
#define FALSE 0
```

Riferimento

La funzione è invocata con un riferimento all'indirizzo della funzione all'interno di una espressione. Questo estende la categoria `<expr>`:

`<expr> ::= ... | <expr1>(<actual_parameter_list>) | ...`

`<expr1>` è una espressione che restituisce un indirizzo di funzione (nel caso più comune `<expr1>` è il nome della funzione, che denota l'indirizzo della funzione stessa). `<actual_parameter_list>` è una lista di espressioni che restituiscono valori di tipo corrispondente ai parametri formali della `<formal_parameter_list>` che compare nella dichiarazione della funzione.

`actual_parameter_list ::= [<expr>{, <expr>}]`

Si osservi che le parentesi quadre `[]` denotano un termine opzionale del BNF e le parentesi graffe `{}` un termine che può essere ripetuto da 0 a un numero arbitrario di volte.

La semantica associata alla chiamata di una funzione, come quella di ogni altra espressione, consiste di un valore restituito e di un insieme di side-effects.

Il valore restituito è il valore restituito dalla espressione che compare nella istruzione `return <expression>` eseguita come ultima istruzione all'interno della funzione.

I side-effects sono quelli che derivano dalla esecuzione dello `<statement>` che compare nella definizione della funzione applicato ai parametri specificati nella chiamata. Tali effetti includono la creazione sullo stack delle variabili corrispondenti ai parametri formali, la loro inizializzazione con i valori dei parametri attuali al momento della invocazione della funzione, la allocazione delle variabili locali alla funzione chiamata e poi la esecuzione dello statement che compone il corpo della funzione chiamata. In particolare, attraverso l'uso dell'operatore di

de-referenziazione, lo statement può modificare variabili esterne allo spazio degli indirizzi della funzione chiamata ma dei quali alla funzione chiamata sia noto l'indirizzo.

Esempio: lo stack

Per esemplificare l'uso delle funzioni consideriamo la rappresentazione in c di uno stack. Non deve essere confuso con lo stack di sistema. È piuttosto una struttura che serve a memorizzare valori su un buffer gestito con politica di accesso Last In First Out. L'esempio offre l'occasione per illustrare la struttura di un programma completo.

```
// inclusioni per l'uso di funzioni di libreria
#include<stdlib.h>
#include<stdio.h>

// definizioni per l'uso dei Booleani
typedef unsigned short int Boolean;
#define TRUE 1
#define FALSE 0

// costante simbolica per il dimensionamento statico del buffer
#define N 128

// prototipi
void main(void);
void init(int* TOS_ptr);
Boolean push(float * buffer, int* TOS_ptr, float value);
Boolean pop(float * buffer, int* TOS_ptr, float* value_ptr);
void getvalue(float * value_ptr);
void putvalue(float value);
void notify_push_failure(void);
void notify_pop_failure(void);
void notify_selection_failure(char selection);

// definizione delle funzioni (da qui al termine)

void main(void)
/* \E un dialogo transazionale nel quale l'utente puo' ripetutamente
   inserire o estrarre numeri, fino a quando richiede la terminazione. */
{
    int TOS;
    float * buffer;
    char selection;
    float value;
    Boolean exit_required;
    Boolean result;
```

```

init(&TOS);

do
{
    exit_required=FALSE;
    selection=getchar();
    switch(selection)
    {
        case 'i':
        case 'I':
            getvalue(&value);
            result=push(buffer,&TOS,value);
            if(result==FALSE)
                notify_push_failure();
            break;
        case 'o':
        case 'O':
            result=pop(buffer,&TOS,&value);
            if(result==TRUE)
                putvalue(value);
            else
                notify_pop_failure();
            break;
        case 'x':
        case 'X':
            exit_required=TRUE;
            break;
        default:
            notify_selection_failure(selection);
    }
}while(exit_required==FALSE);
}

void init(int* TOS_ptr)
/* inizializza a zero il TOS*/
{
    *TOS_ptr=0;
}

Boolean push(float * buffer, int* TOS_ptr, float value)
/* se il buffer non e' pieno inserisce value e restituisce TRUE.
   altrimenti restituisce FALSE */
{
    if(*TOS_ptr<N)
    {
        buffer[*TOS_ptr]=value;
        (*TOS_ptr)++;
        return TRUE;
    }
    else

```

```

    return FALSE;
}

Boolean pop(float * buffer, int* TOS_ptr, float* value_ptr)
/* se il buffer non e' vuoto ne estrae un valore in *value_ptr e
   restituisce TRUE. Altrimenti restituisce FALSE */
{
    if(*TOS_ptr>0)
    {   (*TOS_ptr)--;
        *value_ptr=buffer[*TOS_ptr];
        return TRUE;
    }
    else
        return FALSE;
}

void getvalue(float * value_ptr)
/* acquisisce un float da tastiera */
{
    printf("\n insert value> ");
    scanf("%f", value_ptr);
}

void putvalue(float value)
/* stampa a video un float */
{
    printf("\n extracted value: %f", value);
}

void notify_push_failure(void)
/* notifica il fallimento di un push */
{
    printf("\n buffer full! Insertion cancelled. ");
}

void notify_pop_failure(void)
/* notifica il fallimento di un pop */
{
    printf("\n buffer empty! Extraction impossible. ");
}

void notify_selection_failure(char selection)
/* notifica il fallimento della selezione */
{
    printf("\n %c is not a legal selection. ",selection);
}

```

1.5.7 Dati strutturati

Un dato strutturato è un insieme di variabili, anche di tipo diverso, che possono essere referenziate attraverso un nome collettivo e un indice simbolico. Questo fornisce un meccanismo supportato dal compilatore che garantisce l'associazione tra variabili correlate.

Si pensi ad esempio al caso del descrittore di uno studente, fatto di un nome, un cognome e un numero di matricola; i tre dati possono essere codificati in tre variabili, associate dalla disciplina del programmatore attraverso un buon uso dei nomi e posizionamento conveniente delle dichiarazioni; tuttavia non è difficile pensare al caso in cui dovendo un programma trattare due studenti si possa associare il numero di matricola del primo al nome e cognome del secondo.

Definizione

Mentre le variabili di un array sono posizionate in locazioni contigue e di uguale dimensione, le variabili di un dato strutturato, essendo di tipi diversi, sono posizionate a distanza non regolare. E per ragioni di ottimizzazione della compilazione spesso non sono neppure posizionate in locazioni contigue. Diversamente da quanto avveniva per gli arrays, occorre allora che il compilatore disponga di una *definizione* di tipo dello specifico dato che determini quali sono i campi che lo compongono, quale è la dimensione complessiva del dato, e in quali posizioni relative sono collocati i diversi campi.

La definizione di tipo di un dato strutturato è una nuova categoria sintattica basata sull'uso della parola chiave **struct**:

```
<struct_def> ::= struct <identifier> { { <var_declaration> } };
```

Si noti ancora la differenza tipografica tra i simboli { } che denotano le parentesi di ripetizione BNF, e i simboli { } che denotano invece le parentesi graffe del c.

La definizione di una struttura mette in vita un nuovo tipo, denominato **struct <identifier>**, che è ottenuto componendo i campi con nome e tipo definiti dalla sequenza di dichiarazioni { <var_declaration> } ;:

```
<type> ::= ... | struct <identifier>
```

Dichiarazione

Una volta definito un tipo strutturato, diventa possibile *dichiarare* variabili nel tipo stesso, nello stesso modo con cui si dichiarano variabili nei tipi di base elencati nella categoria **<type>**. Come unica differenza, i tipi strutturati sono riconoscibili per l'uso della parola chiave **struct** che precede il nome del tipo (in realtà un

uso appropriato di `typedef` permetterebbe di denominare il tipo con il solo nome omettendo la parola `struct`).

Il modo con cui sono dichiarate variabili in un tipo strutturato segue la categoria sintattica `<declaration>`, che non è modificata dall'introduzione delle strutture. In particolare, come avviene per tutte le dichiarazioni, il nome associato alla variabile denota la variabile stessa, in questo caso l'insieme dei campi che compongono la struttura.

Riferimento

Una variabile dichiarata in un tipo strutturato e i singoli campi che la compongono possono essere *referenziati* nelle espressioni del programma.

Il riferimento alla variabile strutturata nel suo insieme segue le regole già introdotte nella categoria sintattica `var`.

Il riferimento ai campi della struttura può avvenire in due forme diverse: a partire da un riferimento alla variabile strutturata oppure a partire da una espressione che ne restituisce l'indirizzo. Questo estende la categoria dei riferimenti a variabile `<var>`:

```
<var> ::= ... | <vars>.<identifier> | <exprs>-><identifier> | ...
```

Affinché il riferimento `<vars>.<identifier>` sia legale occorre che `<vars>` sia un riferimento a una variabile di tipo strutturato e `<identifier>` sia l'identificativo di un campo all'interno della definizione del tipo. In tal caso, ovviamente, il riferimento `<vars>.<identifier>` denota il campo `<identifier>` all'interno della variabile denotata da `<vars>`.

In maniera analoga, il riferimento `<exprs>-><identifier>` è legale se `<exprs>` è una espressione che restituisce l'indirizzo di una variabile di tipo strutturato e `<identifier>` è l'identificativo di un campo all'interno della definizione del tipo. In tal caso, il riferimento `<exprs>-><identifier>` denota il campo `<identifier>` all'interno della variabile puntata dall'indirizzo restituito da `<exprs>`.

Esempio: lo Stack

Possiamo adesso riscrivere il codice dello stack in maniera da associare il TOS e il buffer come campi di una stessa struttura. Questo evita che per errore il TOS di uno stack sia usato per indicizzare il buffer di un altro stack, e anche semplifica il passaggio dei parametri raccogliendo sotto un unico puntatore sia il buffer che il TOS. Nel riscrivere il codice, modifichiamo anche la funzione di inizializzazione in modo che essa provveda ad allocare il buffer in base ad un valore di dimensione passato come parametro.

```

...
struct stack{
    int size;
    int TOS;
    float * buffer;
};

...
void main(void)
{
    struct stack stack;
    int size;
    float value;
    Boolean result;
    ...
    getsize(&size);
    init(&stack,size);
    ...
    result=push(&stack,value);
    ...
    result=pop(&stack,&value);
    ...

}

void getsize(int * size_ptr)
/* acquisisce dinamicamente la dimensione dello stack */
{
    printf("\n stack size> ");
    scanf("%d", size_ptr);
}

void init(struct stack * ptr, int size)
/* alloca il buffer e inizializza i campi TOS e size.*/
{
    ptr->size=size;
    ptr->TOS=0;
    ptr->buffer=(float *)malloc(size*sizeof(float));
}

Boolean push(struct stack * ptr, float value)
/* se il buffer non e' pieno inserisce value e restituisce TRUE.
   altrimenti restituisce FALSE. */
{
    if(ptr->TOS<ptr->size)
    {
        (ptr->buffer)[ptr->TOS]=value;
        (ptr->TOS)++;
        return TRUE;
    }
    else

```

```
    return FALSE;
}

Boolean pop(struct stack * ptr, float* value_ptr)
/* se il buffer non e' vuoto ne estrae un valore in *value_ptr e restituisce TRUE.
   altrimenti restituisce FALSE */
{
    if(ptr->TOS>0)
    {
        (ptr->TOS)--;
        *value_ptr=(ptr->buffer)[ptr->TOS];
        return TRUE;
    }
    else
        return FALSE;
}
```

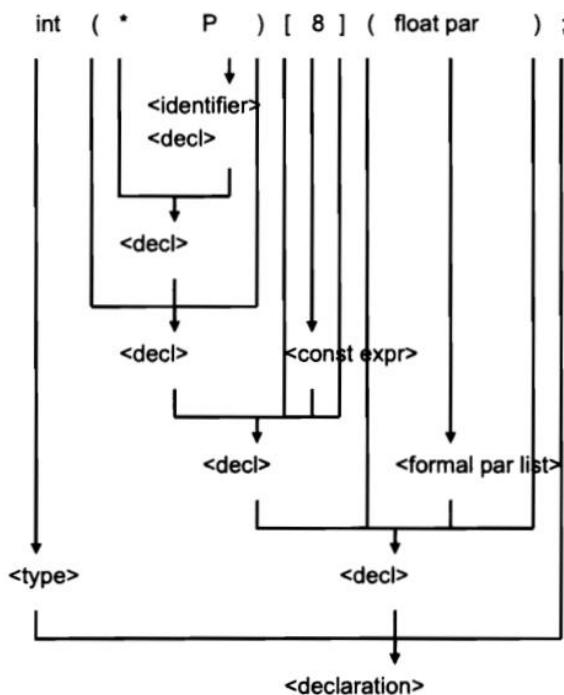
1.6 Esercizi

Es. 1.6.1 Si verifichi la correttezza sintattica e si determini il significato della seguente dichiarazione:

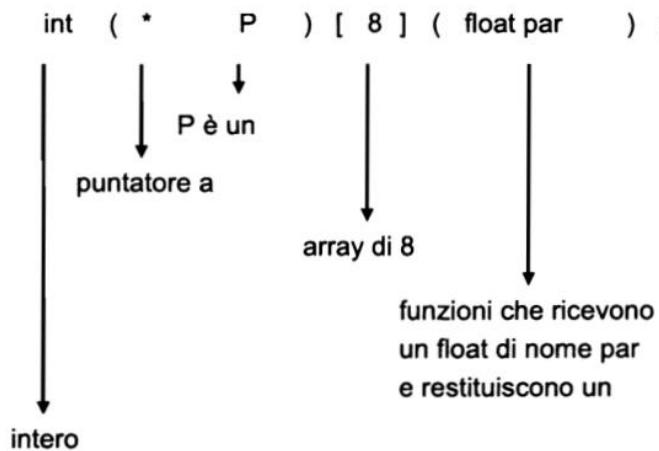
```
int (* P)[8](float par);
```

Soluzione: L'albero sintattico riportato di seguito verifica che `int (* P)[8](float par);` costituisce una dichiarazione in accordo con le riduzioni:

```
declaration ::= type decl ;
decl ::= identifier | *decl | decl[const_expr] | decl(formal_parameter_list) | ...
```



In generale il significato di una dichiarazione consiste nel determinare il nome di un simbolo e il tipo a cui esso è associato. Questo è convenientemente ottenuto leggendo la dichiarazione a partire dal nome e procedendo sempre prima verso destra e poi verso sinistra secondo la regola enunciata nel capitolo 1.5.4. Nel caso specifico questo produce il seguente schema nel quale il significato della dichiarazione è letto procedendo dall'alto verso il basso “P è un puntatore ad array di funzioni che ricevono un float e restituiscono un intero”:



□

Es. 1.6.2 Si consideri il riferimento a variabile:

`(A[count].ptr)->codice`

assumendo che esso compaia nel contesto delle seguenti definizioni e dichiarazioni:

```

struct record A[128];
int count;

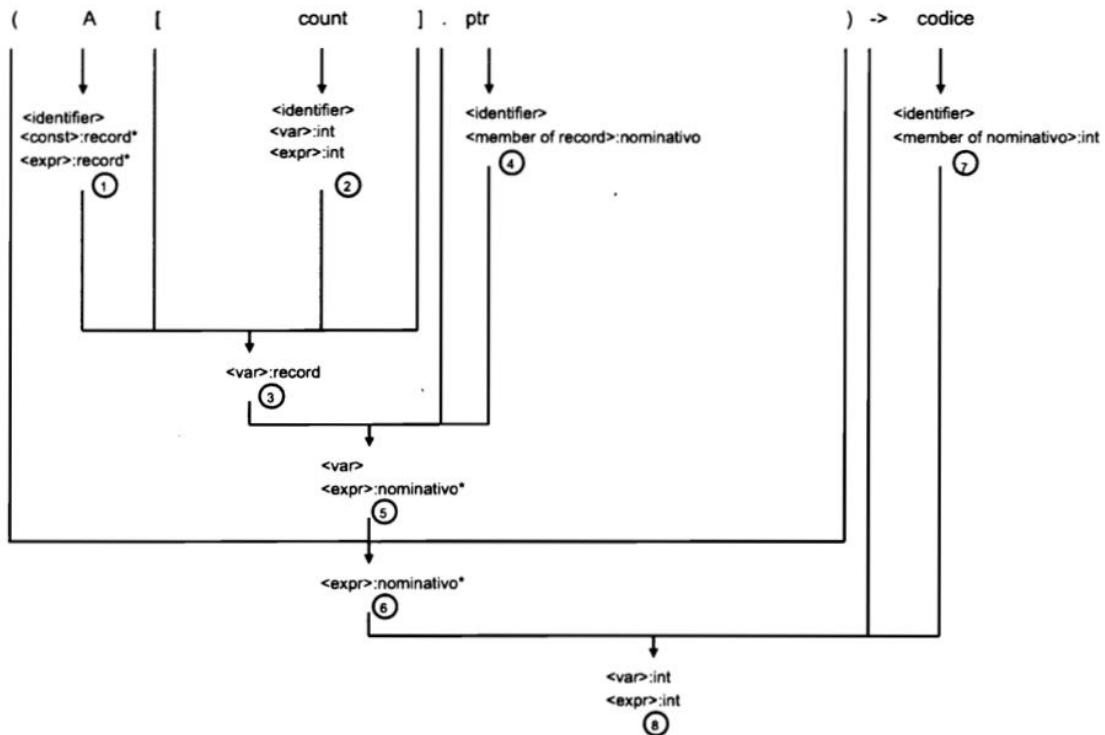
struct record{
    ...
    struct nominativo *ptr;
};

struct nominativo{
    ...
    int codice;
};

```

Si costruisca l'albero sintattico che riduce il riferimento, evidenziando il frammento di sintassi del linguaggio c coinvolto e il significato derivato nell'interpretazione.

Soluzione:



```

var ::= identifier | expr_addr[expr_int] | (var) |
        var_struct.identifier | expr_structaddr->identifier|...
expr ::= const|var|(expr)|...
    
```

In generale il significato di un riferimento consiste nell'identificare un simbolo altrove dichiarato. Nel caso specifico: 1: A è una espressione (costante) che restituisce l'indirizzo di base di un array di variabili di tipo record. 2: count è un riferimento a variabile di tipo intero e quindi anche un'espressione che restituisce il valore della variabile stessa. 3: A[count] è un riferimento alla variabile di tipo record all'offset count a partire dall'indirizzo A. 4: ptr è un membro di tipo puntatore a nominativo nella definizione del tipo strutturato record. 5: A[count].ptr è un riferimento a una variabile di tipo puntatore a nominativo. 6: (A[count].ptr) è un'espressione che restituisce lo stesso valore dell'espressione A[count].ptr. 7: codice è un membro di tipo puntatore a nominativo nella definizione del tipo strutturato record. 8: (A[count].ptr)->codice è un riferimento al campo intero codice nella variabile strutturata di tipo nominativo puntata da A[count].ptr. □

Es. 1.6.3 Si consideri l'espressione:

```
ptrptr=&((*ptrptr)->next_ptr)
```

assumendo che essa compaia nel contesto di un programma nel quale figurano le seguenti dichiarazione di variabile e definizione di struttura:

```

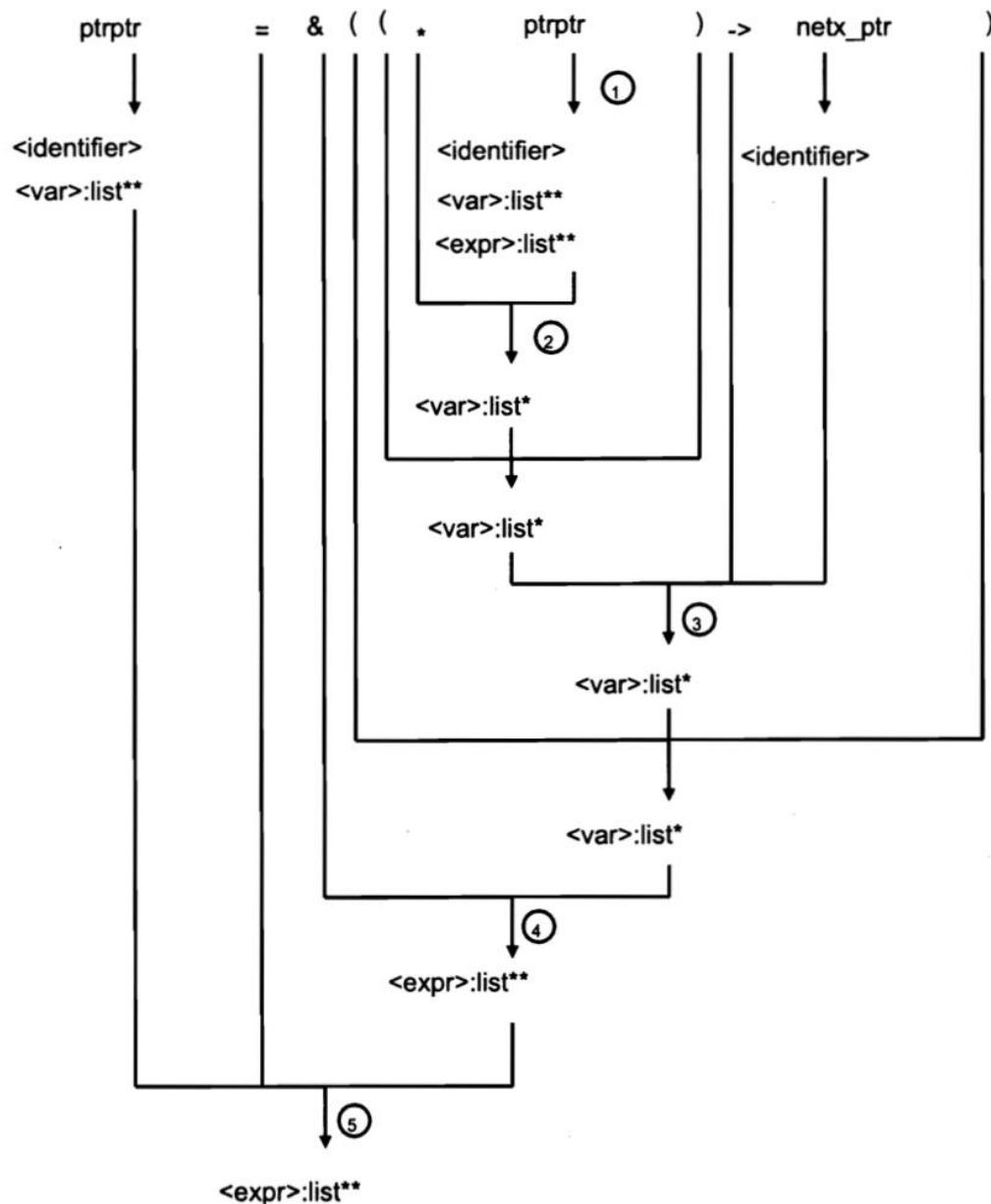
struct list{
    ...
    struct list * next_ptr;
};

struct list * ptrptr;

```

Si definiscano sintassi e semantica dell'espressione, costruendone l'albero sintattico e elencando le produzioni del frammento di linguaggio c coinvolte nella sua interpretazione.

Soluzione:



```

var ::= identifier | expr_addr[expr_int] | (var) |
      *expr_addr | expr_structaddr->identifier|...

```

```
expr ::= var | (expr) | var=expr ...
```

In generale, il significato di un'espressione consiste di un valore restituito (in un qualche tipo) e di un insieme di side-effects sulle variabili. Nel caso specifico: 1: `ptrptr` è una variabile di tipo puntatore a indirizzo di lista, e quindi anche una espressione senza side effects che restituisce quello stesso indirizzo. 2: `*ptrptr` è la variabile di tipo indirizzo di lista puntata da `ptrptr`; il suo contenuto indica una variabile di tipo `list` composta di un campo `value` di tipo `float` e un campo `next_ptr` di tipo puntatore a lista. 3: `(*ptrptr)->next_ptr` è il campo `next_ptr` nella variabile di tipo lista puntata da `(*ptrptr)`. 4: `&((*ptrptr)->next_ptr)` è un'espressione che non produce side-effects e restituisce l'indirizzo della variabile del punto 3. 5: `ptrptr=&((*ptrptr)->next_ptr)` è un'espressione che restituisce lo stesso valore restituito dall'espressione del punto 4 e lo assegna alla variabile `ptrptr`. □

Es. 1.6.4 Si definisca la funzione `c` che riceve da tastiera valori interi fino a quando il prodotto degli ultimi due valori inseriti non è uguale al valore del terzultimo valore inserito.

Soluzione: Gli ultimi tre valori letti sono rappresentati in tre variabili `old`, `middle` e `recent`. Ad ogni iterazione si spostano `middle` su `old`, `recent` su `middle`, e si legge un nuovo valore in `recent`. Si osservi che prima di avviare la verifica occorre avere letto almeno tre dati. L'iterazione è convenientemente controllata con un `do-while` in quanto il controllo di terminazione viene effettuato dopo la azione di lettura:

```
void trap(void)
{ int old;
  int middle;
  int recent;

  scanf("%d",&middle);
  scanf("%d",&recent);
  do{
    old=middle;
    middle=recent;
    scanf("%d",&recent);
  }while(recent*middle!=old);
}
```

□

Es. 1.6.5 Si definisca la funzione c che riceve come parametro un valore N e poi acquisisce numeri interi da tastiera fino a che gli ultimi N numeri inseriti sono tra loro uguali.

Soluzione: Viene estesa la soluzione dell'esercizio 1.6.4 rappresentando i dati su un vettore anziché in variabili indipendenti. Anziché spostare N-1 valori ad ogni nuova lettura, viene fatto ruotare un indice count il quale indica sempre la posizione nella quale è contenuto il valore più vecchio, e quindi nella quale deve essere letto il prossimo valore. Ad ogni lettura, l'indice è incrementato con una operazione a modulo N. La fase di caricamento iniziale del buffer è realizzata con una iterazione (for), mentre il test di terminazione è implementato in una sotto-funzione:

```
void trap(int N)
{   int count;
    int * V;

    V=(int *)malloc(N*sizeof(int));
    for(count=0;count<N-1;count++)
        scanf("%d",&V[count]);

    count=N-1;
    do{
        scanf("%d",&V[count]);
        count=(count+1)%N;
    }while(is_constant(V,N)==TRUE);

    free(V);
}

Boolean is_constant(int * V, int size)
/* Restituisce true se tutti gli elementi del vettore V di dimensione size
   sono uguali. Assume size>0 */
{
    int count;
    Boolean no_mismatch_found;

    count=1;
    no_mismatch_found=TRUE;
    while(no_mismatch_found==TRUE&& count<N)
        if(V[0]==V[count])
            count++;
        else
            no_mismatch_found=TRUE;

    return no_mismatch_found;
}
```

Per un'implementazione più efficiente, sia in termini di spazio occupato che di operazioni effettuate, è possibile osservare che per verificare che gli ultimi N valori inseriti sono identici non è necessario averli rappresentati su un vettore, ma è invece sufficiente mantenere due variabili che rappresentano l'ultimo valore inserito e il numero di ripetizioni consecutive che esso ha avuto:

```
void trap2(int N)
{  int count;
   int value;
   int new_value;

   scanf("%d",value);
   count=1;
   do{
      scanf("%d",&new_value);
      if(new_value==value)
      {   count++;
      }
      else
      {   value=new_value;
          count=1;
      }
   }while(count<N);
}
```

□

Es. 1.6.6 Si definisca la funzione *c* che riceve due valori interi *N* e *Sum* e poi acquisisce numeri interi da tastiera fino a che la somma degli ultimi *N* numeri inseriti è uguale a *Sum*.

Soluzione: Il problema può essere ricondotto al caso dell'esercizio 1.6.5 e trattato con la funzione *trap()* a meno di sostituire l'invocazione di *is_constant()* con una funzione *sum_equals_value()* che verifica se la somma dei valori sul vettore *V* è pari al valore *sum*.

In una implementazione computazionalmente più efficiente è però possibile evitare di dovere ricalcolare la somma ad ogni nuovo valore inserito:

```
void trap_until_sum_over_N(int N, int Sum)
/* alloca un vettore di dimensione N, ci legge i
   primi N-1 elementi calcolandone la somma in _Sum.
   Poi ripetutamente legge un valore in una posizione che ruota a partire da N-1.
   Ad ogni lettura aggiorna _Sum scaricando il valore che viene sovrascritto e
   caricando il nuovo valore letto. */
{
   int count;
   int * V;
```

```

int _Sum

V=(int *)malloc(N*sizeof(int));

for(count=0, _Sum=0;count<N-1;count++)
{   scanf("%d",&V[count]);
    _Sum+=V[count];
}

_Sum+=V[N-1];
count=N-1;
do{
    _Sum-=V[count];
    scanf("%d",&V[count]);
    _Sum+=V[count];
    count=(count+1)%N;
}while(_Sum!=Sum);

free(V);
}

```



Esercizio 1.6.7 Si definisca la funzione *c* che riceve in ingresso due vettori di numeri interi *A1* e *A2*, di lunghezza *N1* e *N2* rispettivamente, e verifica se *A2* e' contenuto in *A1*.

Soluzione: Un indice base scandisce il vettore *A1*. Per ogni posizione di base, una scansione verifica se *A1*[base+offset]==*A2*[offset] per i valori di offset da 0 a *N2*. In caso affermativo la ricerca termina con successo, altrimenti viene incrementato base. La ricerca termina con un fallimento se base eccede la posizione *N1-N2*. Si osservi che le due scansioni fanno uso di due variabili Booleane distinte denominate *found* e *mismatch* che codificano due diverse condizioni logiche.

```

Boolean match(int *A1, int N1, int * A2, int N2)
{
    int base;
    int offset;
    Boolean found;
    Boolean mismatch;

    found=FALSE;
    base=0;
    while(found==FALSE && base+N2<N1)
    {   mismatch=FALSE;
        offset=0;

```

```

        while(mismatch==FALSE&&offset<N2)
        {
            if(A1[offset]!=A2[base+offset])
                mismatch=TRUE;
            else
                offset++;
        }
        if(mismatch==FALSE)
            found=TRUE;
        else
            base++;
    }
    return found;
}

```

□

Es. 1.6.8 Si definisca la funzione *c* che riceve in ingresso due indici *n* e *l*, e due matrici *A* e *B* di dimensione *NxM* e *MxL* rispettivamente, e restituisce in uscita il valore dell'elemento *nl* della matrice prodotto *A*B*.

Nota: l'elemento *nl* del prodotto *A * B* esiste sse $0 \leq n < N$ e $0 \leq l < L$, e in tal caso esso vale $[A * B]_{nl} = \sum_{m=0}^{M-1} A_{nm}B_{ml}$ (prodotto scalare della riga *n* di *A* per la colonna *l* di *B*).

Soluzione: Le due matrici sono rappresentate su array monodimensionali. La funzione assume la responsabilità di verificare la legalità degli indici *n* e *l*.

```

Boolean riga_colonna(float * A, float * B, int N, int M, int L, int n, int l,
                      float * value_ptr)

{   int m;

    if(n<N&&n>=0 && l<L&&l>=0)
    {
        for(m=0, *value_ptr=0; m<M; m++)
            *value+=A[n*M+m]*B[m*L+l];
        return TRUE;
    }else
    {
        return FALSE;
    }
}

```

□

Es. 1.6.9 Si definisca la funzione *c* che riceve in ingresso due matrici di valori float *A* e *B* di dimensione *NxM* e *MxL* rispettivamente, e alloca e restituisce in uscita la matrice prodotto *C* di dimensione *NxL*.

Soluzione: La soluzione estende quanto già sviluppato nel precedente esercizio 1.6.8. Si osservi l'uso del doppio puntatore `**C_ptr` necessario per restituire l'indirizzo al quale `malloc()` alloca l'array su cui è calcolato il prodotto delle due matrici di ingresso.

```
void prod(float * A, float * B, int N, int M, int L, float **C_ptr)
{ int n;
  int m;
  int l;

  *C_ptr=(float *)malloc(N*L*sizeof(float));
  for(n=0;n<N;n++)
  {   for(l=0;l<L;l++)
      {   for(m=0,*C_ptr[n*L+l]=0;m<M;m++)
          {   *C_ptr[n*L+l]+=A[n*M+m]*B[m*L+l];
          }
      }
  }
}
```

□

Es. 1.6.10 Si definisca la funzione `c` che riceve in ingresso una matrice $A_{N \times N}$, un vettore X_N e uno scalare l e verifica se X è autovettore di A con autovalore l . Si assuma che A , X e l , siano valori float e se ne tenga conto rispetto al problema della precisione finita.

Nota: X è autovettore di A con autovalore l sse risulta $(A - l \cdot Id)X = 0$ dove Id è la matrice di identità, ovvero sse $\forall n \in [0, N-1] \quad \sum_{m=0}^{M-1} A_{nm}X_m = lX_n$

Soluzione: Il calcolo matriciale non aggiunge niente rispetto a quanto già visto negli esercizi precedenti. La verifica dell'identità $\sum_{m=0}^{M-1} A_{nm}X_m = lX_n$ incontra il problema della precisione finita sui numeri float, per cui l'operatore di uguaglianza `==` è sostituito da una funzione `is_equal()` che restituisce TRUE quando la differenza tra i due termini comparati è minore della precisione di macchina. La precisione è calcolata in termini relativi.

```
typedef int Boolean;
#define TRUE 1
#define FALSE 0

Boolean isEigenvalue(float * A, float * X, float lambda, int N, float epsilon)
/* epsilon denota la precisione relativa con la quale viene testata
   l'identità sui float. */
{
```

```

int n;
Boolean no_mismatch_found;

no_mismatch_found=TRUE;
n=0;
while(no_mismatch_found && n<N)
{   for(m=0,sum=0; m<N-1; m++)
    sum+=A[n*N+m]*X[m];
    if(is_equal(sum,lambda*X[n], epsilon)==FALSE)
        no_mismatch_found=FALSE;
}
return no_mismatch_found;
}

Boolean is_equal(float a, float b, float epsilon)
/* restituisce TRUE se l'errore relativo nella approssimazione
di a con b e' minore di epsilon */
{
    float diff_rel;

    diff_rel=(a-b)/a;
    if(diff_rel<epsilon && diff_rel>-epsilon)
        return TRUE;
    else
        return FALSE;
}

```



Es. 1.6.11 Assegnati una matrice di numeri reali A di dimensione $N \times N$, un vettore di numeri reali V di dimensione N , e un vettore di indici $Index$ di dimensione $M \leq N$ i cui elementi prendono valore in $[0, N - 1]$, denotiamo con A_{Index} la matrice ottenuta selezionando in A le righe e le colonne con gli indici contenuti in $Index$ (ovvero $A_{Index}[i, j] = A[Index[i], Index[j]]$). Analogamente denotiamo con V_{Index} il vettore ottenuto selezionando in V le righe con gli indici contenuti in $Index$ (ovvero $V_{Index}[i] = V[Index[i]]$).

Si definisca la funzione c che riceve in ingresso una rappresentazione della matrice A , del vettore V , e del vettore $Index$, e che provvede ad allocare e calcolare una rappresentazione del vettore di dimensione M definito come $W = A_{Index} * V_{Index}$.

Nota: per aiutare la comprensione del problema, consideriamo il caso di esempio

in cui $N=5$, $M=3$ e:

$$A = \begin{pmatrix} 1. & 2. & 3. & 5. & 7. \\ 2. & 3. & 5. & 7. & 11. \\ 3. & 5. & 7. & 11. & 13. \\ 5. & 7. & 11. & 13. & 17. \\ 7. & 11. & 13. & 17. & 19. \end{pmatrix} V = \begin{pmatrix} 1. \\ 2. \\ 4. \\ 8. \\ 16. \end{pmatrix} Index = \begin{pmatrix} 0 \\ 2 \\ 3 \end{pmatrix} \quad (1.11)$$

In tale caso: A_Index è la matrice 3×3 ottenuta selezionando in A le righe 0, 2 e 3; V_Index è il vettore di dimensione 3×1 ottenuto selezionando in V le righe 0, 2 e 3, e W è il vettore di dimensione 3 ottenuto dal loro prodotto (calcolato con lo schema riga colonna):

$$A_Index = \begin{pmatrix} 1. & 3. & 5. \\ 3. & 7. & 11. \\ 5. & 11. & 13. \end{pmatrix} V_index = \begin{pmatrix} 1. \\ 4. \\ 8. \end{pmatrix} W = \begin{pmatrix} 52. \\ 119. \\ 153. \end{pmatrix} \quad (1.12)$$

Soluzione: Assumiamo come usuale che vettori e matrici siano rappresentati su arrays monodimensionali.

Una soluzione diretta del problema può essere quella di allocare due ulteriori arrays di dimensione $M \times M$ e M , costruirsi gli arrays A_Index e V_Index copiando i valori selezionati in A e V attraverso il vettore Index, e poi calcolare W come prodotto:

```
void prod_restricted(float * A, float * V, int * Index,
                      int N, int M, float ** Wptr)
/* la funzione assume che sia già verificato che M<=N-1.
   Assume la responsabilità di allocazione dell'array *W_ptr.
   Non gestisce il caso di errore di un fallimento nell'allocation.*/
{
    float * V_Index;
    float * A_Index;
    int row;
    int col;

    *Wptr=(float *)malloc(sizeof(float)*M);

    V_Index=(float *)malloc(sizeof(float)*M);
    for(row=0; row<M; row++)
        V_Index[row]=V[Index[row]];

    A_Index=(float *)malloc(sizeof(float)*M*M);
    for(row=0; row<M; row++)
        for(col=0; col<M; col++)
            A_Index[row*M+col]=A[Index[row]*N+Index[col]];

    for(row=0; row<M; row++)

```

```

    for(col=0, *Wptr[row]=0;col<M;col++)
        *Wptr[row]+=A_Index[row*N+col]*V_Index[col];

    free(V_index);
    free(A_index);
}

```

In una implementazione più raffinata è possibile evitare l'allocazione e la copia, derivando W direttamente dagli arrays A e V e utilizzando il vettore Index come indice di accesso:

```

void prod_restricted(float * A, float * V, int * Index,
                      int N, int M, float ** Wptr)
/* la funzione assume che sia già verificato che M≤N-1
   Assume la responsabilità di allocazione dell'array *Wptr.
   Non gestisce il caso di errore di un fallimento nell'allocation. */
{
    int row;
    int col;

    *Wptr=(float *)malloc(sizeof(float)*M);

    for(row=0;row<M;row++)
        for(col=0, *Wptr[row]=0;col<M;col++)
            *Wptr[row]+=A[Index[row]*N+Index[col]]*V[Index[col]];
}

```



Capitolo 2

Strutture dati e algoritmi elementari

Introduciamo in questo capitolo i concetti di struttura dati e di algoritmo trattando prima le liste e gli alberi binari e poi i problemi della ricerca e dell'ordinamento.

La trattazione fornisce l'opportunità per applicare in modo anche raffinato i costrutti del c già introdotti nella prima parte del testo. Permette anche di introdurre in maniera concreta alcuni concetti fondamentali della programmazione: la separazione tra logica e implementazione di una struttura dati; i concetti di ricorsione e iterazione; la valutazione della complessità di un algoritmo e di un problema; la verifica della correttezza; la disciplina nella programmazione e il riuso di schemi e soluzioni.

2.1 Liste

Una lista è una successione finita di valori di un tipo. L'informazione che questo codifica consiste di due componenti: l'insieme dei valori e la relazione di ordine tra i valori stessi. In questo senso, $\{1, 5, 2\}$ è una lista diversa da $\{1, 2, 5\}$.

Come ogni tipo di dato, la lista è qualificata non solo dai valori che rappresenta, ma anche dalle operazioni che ci si eseguono sopra. Queste sono l'inserimento e la cancellazione, la visita e la ricerca, l'inizializzazione.

- L'inserimento aggiunge un nuovo valore alla lista. In diverse varianti dell'operazione, il valore può essere inserito in testa, in coda, o in una posizione intermedia determinata da un qualche criterio di posizionamento (ad esempio nella creazione di una lista ordinata).
- La cancellazione rimuove un elemento dalla lista. Anche in questo caso, diverse varianti operano sulla testa, la coda, oppure su un elemento determinato in base a un qualche criterio (ad esempio il valore che contiene) o possono restituire il valore che era memorizzato sull'elemento che è stato cancellato.
- La visita applica un trattamento comune a tutti gli elementi della lista. Una visita può stampare i valori della lista, calcolarne la somma o identificarne il massimo, modificarne il valore secondo una comune legge di trasformazione.
- La ricerca determina se la lista contiene un elemento qualificato da una condizione sul valore. Tipicamente restituisce vero/falso, ma può anche restituire l'indirizzo o comunque la posizione dell'elemento cercato nel caso in cui questo sia presente.
- L'inizializzazione non ha un significato proprio nell'elaborazione dell'informazione. È strumentale alla particolare rappresentazione concreta con cui viene codificata la lista. Tipicamente serve ad asserire un qualche invarianto che viene sfruttato e mantenuto nel corso dell'elaborazione. Spesso serve a identificare il primo e/o l'ultimo elemento della lista.

Esistono due forme diverse con cui è possibile rappresentare la lista: la rappresentazione sequenziale e quella collegata. Le differenze tra le due attengono sostanzialmente al modo con cui è rappresentata la relazione d'ordine tra gli elementi della lista.

- Nella *rappresentazione in forma sequenziale*, i valori della lista sono rappresentati su un vettore e il loro ordine è codificato in maniera implicita dalla posizione nel vettore.

- Nella *rappresentazione in forma collegata*, la relazione d'ordine è codificata in modo esplicito associando ciascun valore ad una informazione che permette di identificare il successore. Di questa forma esistono due ulteriori varianti che si differenziano a seconda che l'informazione sul successore sia codificata usando gli *indici* di un vettore o i *puntatori* a un insieme di variabili allocate indipendentemente.

2.1.1 Rappresentazione in forma sequenziale

In prima approssimazione, una rappresentazione in forma sequenziale può essere ottenuta codificando i valori della lista su un array associato a una variabile che indica il numero di elementi rappresentati.

Questo schema di rappresentazione, che coincide sostanzialmente con lo stack già discusso nei capitoli 1.5.6 e 1.5.7, permette di eseguire efficientemente le operazioni di inserimento ed estrazione in coda, la visita e la ricerca.

È però problematica la realizzazione di un inserimento o una estrazione in testa che richiedono una catena di copie di complessità proporzionale al numero di elementi memorizzati: per inserire un elemento in testa, l'elemento in posizione N viene copiato in posizione $N + 1$, l'elemento in posizione $N - 1$ viene copiato in posizione N , e così via fino a che l'elemento in posizione 1 viene copiato in posizione 2; a questo punto la posizione 1 è libera per accettare il nuovo valore inserito.

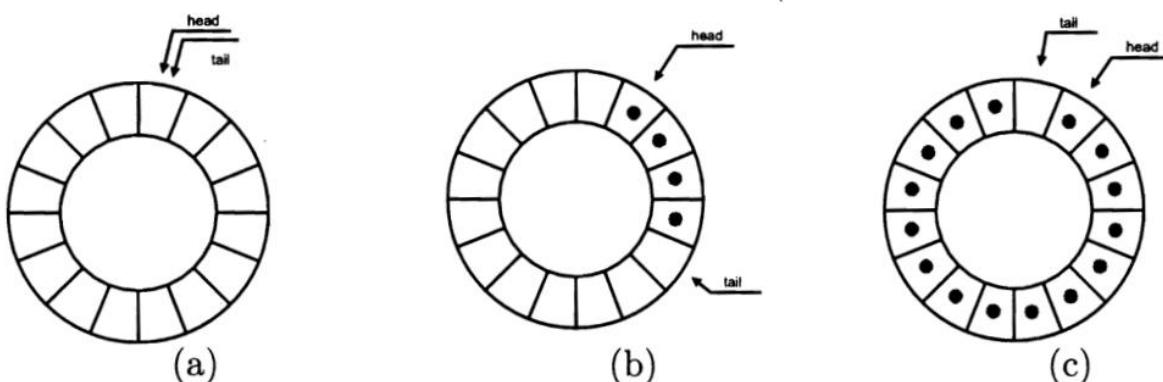


Figura 2.1: Lo schema del buffer circolare: **a)** il buffer è vuoto se e solo se **head==tail**; **b)** se il buffer è non vuoto, i valori sono memorizzati in senso orario dalla posizione **head** alla posizione **tail-1** inclusive; **c)** quando $(tail+1)\%size==head$ il buffer non può accettare un ulteriore elemento perché la condizione raggiunta non sarebbe distinguibile dal caso del buffer vuoto.

Il problema viene superato organizzando l'array nello schema di un buffer circolare (vedi Fig.2.1). Concettualmente questo realizza un'area di memoria circolare controllata con due indici **head** e **tail** che realizzano il seguente invarianto: se

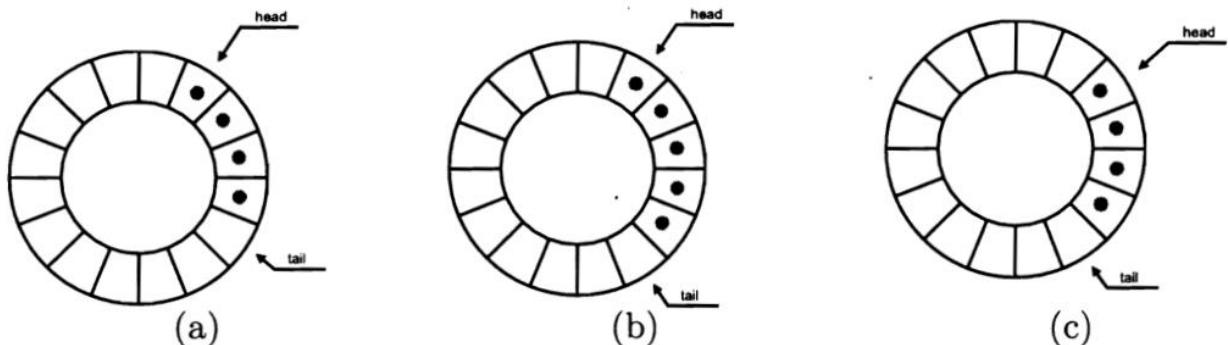


Figura 2.2: Lo schema del buffer circolare: le operazioni di inserimento in coda (**a**→ **b**) e di cancellazione in testa (**b**→ **c**) fanno scorrere in senso orario il segmento di buffer occupato.

`head==tail` la lista è vuota; altrimenti la locazione `head` contiene la testa, `tail-1` contiene la coda, e i valori della lista sono memorizzati nel loro ordine nel segmento compreso tra gli indici da `head` incluso a `tail` escluso. La lista è considerata piena quando `head==tail+1`, anche se in tale condizione una locazione è in effetti vuota. Se anche quella locazione venisse riempita risulterebbe `head==tail`, rendendo non distinguibile le condizioni di lista piena e lista vuota.

Le operazioni di cancellazione/inserimento in testa spostano avanti/indietro l'indice `head`, quelle in coda hanno lo stesso effetto sull'indice `tail`. Combinando le diverse operazioni il segmento occupato dalla lista ruota nel buffer, allungandosi in senso orario/antiorario per gli inserimenti in testa/coda e accorciandosi in senso antiorario/orario per le cancellazioni in testa/coda (vedi Fig.2.2).

Lo schema concettuale è concretamente implementato su un array lineare applicando agli indici `first` e `last` un'algebra a modulo pari alla dimensione dell'array, che è convenientemente realizzato con l'operatore `%` del linguaggio C.

```

struct list{
    float * buffer;
    int size;
    int head;
    int tail;
};

void init(struct list * ptr, int size)
/* Alloca il buffer su cui sono memorizzati i valori della lista e
   asserisce l'invariante della rappresentazione:
   se head==tail la lista e' vuota;
   altrimenti head e' l'indice dell'elemento di testa e
   tail e' l'indice della prima posizione successiva all'elemento di coda
*/
{
    ptr->buffer=(float *)malloc(size*sizeof(float));
}

```

```

ptr->size=size;
ptr->head=0;
ptr->tail=0;
}

Boolean suf_insert(struct list * ptr, float value)
/* Inserimento in coda: aumenta tail */
{
    if((ptr->tail+1)%ptr->size)!=ptr->head)
    {
        ptr->buffer[ptr->tail]=value;
        ptr->tail=(ptr->tail+1)%ptr->size;
        return TRUE;
    }
    else
        return FALSE;
}

Boolean pre_insert(struct list * ptr, float value)
/* Inserimento in testa: riduce head */
{
    if((ptr->tail+1)%ptr->size)!=ptr->head)
    {
        ptr->head=(ptr->head+ptr->size-1)%ptr->size;
        ptr->buffer[ptr->head]=value;
        return TRUE;
    }
    else
        return FALSE;
}

void visit(struct list * ptr)
/* Visita di stampa. */
{
    int position;
    for(position=ptr->head; position!=ptr->tail; position=(position+1)%ptr->size)
        printf("%f",ptr->buffer[position]);
}

Boolean search(struct list * ptr, float value)
/* Restituisce TRUE se value e' contenuto nella lista, FALSE altrimenti */
{
    Boolean found;
    int position;

    position=ptr->head;
    found=FALSE;
    while(found==FALSE && position!=ptr->tail )
    {   if(ptr->buffer[position]==value)

```

```

        found=TRUE;
    else
        position=(position+1)%ptr->size;
    }
    return found;
}

```

Si osservi che le operazioni di inserimento e cancellazione restituiscono un Booleano per trattare le eccezioni in cui la lista è piena o vuota, rispettivamente; nel corpo della funzione questo risulta in una guardia di ingresso che distingue il caso comune da quello di eccezione. Come buona pratica, il caso comune è programmato prima dell'eccezione.

Si osservi nell'operazione di inserimento in testa che per decrementare `head` di 1 si usa l'espressione `head=(head+size-1)%size` che include senza eccezione il caso in cui `head` vale 0.

Si osservi infine che sulle operazioni di visita e di ricerca potrebbe essere passata la lista piuttosto che il suo indirizzo (i.e. il prototipo della funzione potrebbe avere la forma `void visit(struct list);`) visto che l'operazione non modifica il valore di alcun membro della struttura. Nella implementazione, si è invece passato l'indirizzo per la ragione dell'efficienza, in modo da ridurre il numero dei parametri copiati sullo stack.

La lista in forma sequenziale è inadeguata laddove siano richieste operazioni di inserimento o cancellazione in posizione intermedia. Tali operazioni possono in effetti essere implementate, ma richiedono una catena di operazioni di copia di *costo lineare*, ovvero proporzionale al numero di elementi nella lista.

La lista sequenziale ha anche il limite di richiedere un dimensionamento statico del buffer, che impedisce di adattare la memoria allocata al numero di elementi effettivamente memorizzati nel corso della computazione. Questo diventa un problema non superabile laddove non sia a priori noto un limite superiore sul numero di elementi che la lista può contenere.

La realizzazione in forma sequenziale ha però il pregio, non offerto dalle liste in forma collegata, di permettere l'accesso diretto ai singoli elementi. Questo ad esempio permette di selezionare in tempo costante l'elemento in posizione mediana. Come vantaggio minore, la forma sequenziale ha anche la capacità di permettere in modo naturale la visita in ordine inverso.

2.1.2 Rappresentazione collegata con arrays e indici

La rappresentazione collegata con arrays e indici memorizza ancora i valori su un buffer ma virtualizza la relazione di sequenza dei valori. Il buffer è un array di records, ciascuno dei quali contiene un valore e l'indice del record del buffer che

contiene il successivo valore della lista. L'elemento di coda ha nell'indice un valore illegale che non corrisponde ad alcun record nel buffer; tipicamente il valore illegale è la dimensione del buffer stesso (vedi Fig.2.3).

I records non usati sono anche essi concatenati attraverso gli indici in maniera da formare una lista dei records disponibili. Questa permette di eseguire con un numero costante di operazioni la ricerca di un record libero. A sua volta, questo risulta decisivo per eseguire l'inserimento con un numero di operazioni costante indipendente dalla dimensione della lista e dal numero di valori che essa contiene.

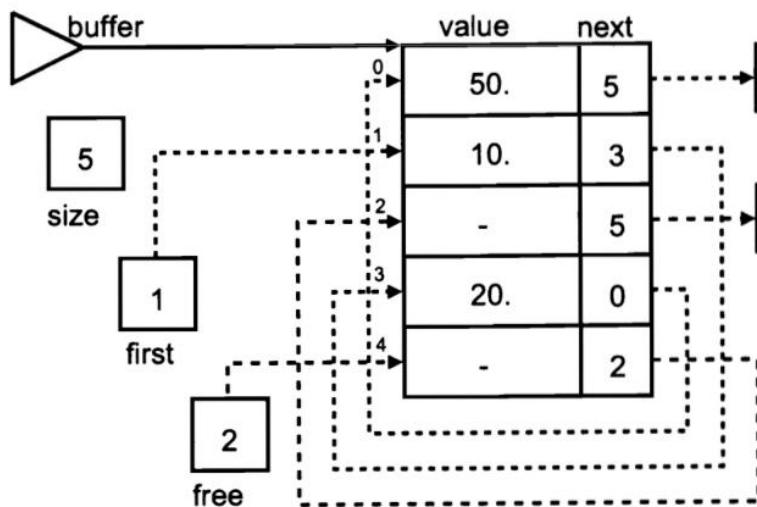


Figura 2.3: La rappresentazione della lista $\{10., 20., 50.\}$ in forma collegata con indici su un buffer di dimensione 5. I records 2 e 4 sono concatenati sulla lista degli elementi liberi.

```

struct record{
    float value;
    int next;
};

struct list{
    int first;
    int free;
    int size;
    struct record * buffer;
};

void init(struct list * ptr, int size)
/* alloca il buffer e asserisce l'invariante della lista:
   first ha il valore illegale che indica che non ci sono elementi memorizzati;
   free indirizza il primo record;
   tutti i records sono concatenati a partire dal primo

```

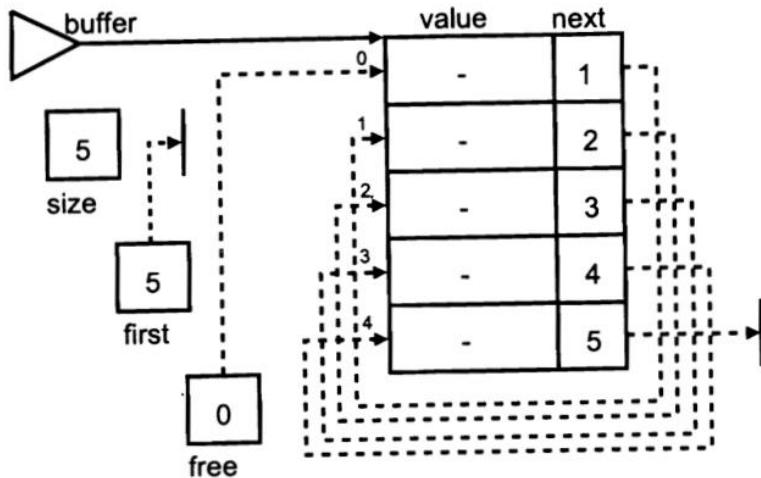


Figura 2.4: Configurazione all'inizializzazione della lista in forma collegata con indici: `first` contiene l'indice illegale; tutti i records del buffer sono concatenati sulla lista degli elementi liberi che origina dall'indice `free`.

```

per averli disponibili come records liberi.
*/
{
    int count;

    ptr->buffer=(struct record *)malloc(size*sizeof(struct record));
    ptr->size=size;
    ptr->first=ptr->size;
    ptr->free=0;
    for(count=0;count<ptr->size;count++)
        ptr->buffer[count].next=count+1;
}

void visit(struct list * ptr)
{
    int position;

    position=ptr->first;
    while(position!=ptr->size)
    {   printf("%f",ptr->buffer[position].value);
        position=ptr->buffer[position].next;
    }
}

Boolean search(struct list * ptr, float value)
{
    int position;
    Boolean found;

```

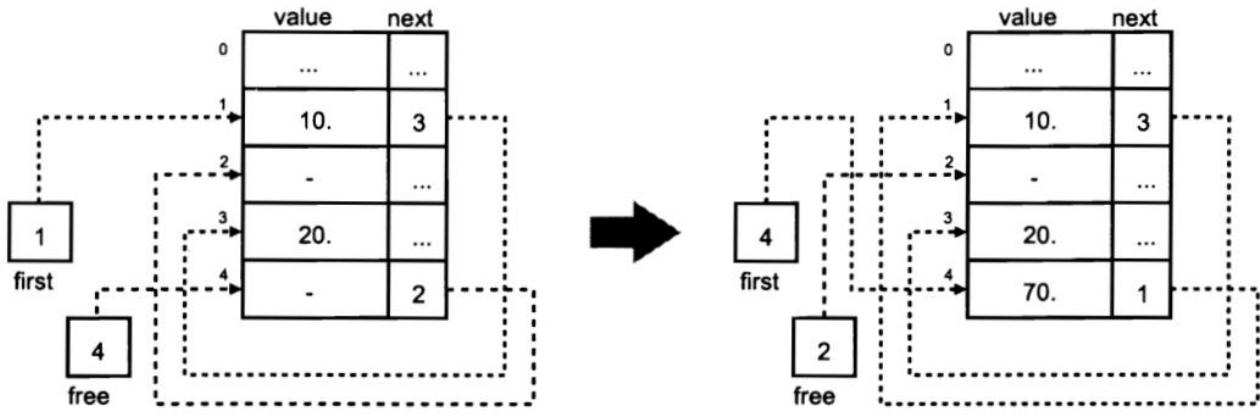


Figura 2.5: L'operazione di inserimento in testa su una lista in forma collegata con indici: il record in testa alla lista degli elementi liberi viene portato sulla lista degli elementi utili e riceve il nuovo valore inserito (70. nell'esempio).

```

position=ptr->first;
found=FALSE;
while(found==FALSE && position!=ptr->size)
    if(ptr->buffer[position].value==value)
        found=TRUE;
    else
        position=ptr->buffer[position].next;
}

Boolean pre_insert(struct list * ptr, float value)
/* se la lista libera non \e vuota, ne sgancia il primo elemento,
   lo aggancia in testa alla lista dei valori
   e ci memorizza il valore da inserire */
{
    int moved;

    if(ptr->free!=ptr->size)
    {
        moved=ptr->free;
        ptr->free=((ptr->buffer)[ptr->free]).next;

        ptr->buffer[moved].value=value;
        ptr->buffer[moved].next=ptr->first;
        ptr->first=moved;

        return TRUE;
    }else
        return FALSE;
}

```

```

Boolean suf_insert(struct list * ptr, float value)
/* Se la lista libera non e' vuota, ne sgancia il primo elemento
e lo aggancia in coda alla lista dei valori,
ci copia sopra il valore da inserire
e attribuisce il valore illegale al successore.
Altrimenti restituisce FALSE. */
{
    int moved;
    int * position_ptr;

    if(ptr->free!=ptr->size)
    {
        moved=ptr->free;
        ptr->free=((ptr->buffer)[ptr->free]).next;

        position_ptr=&ptr->first;
        while(*position_ptr!=ptr->size)
            position_ptr=&(ptr->buffer[*position_ptr].next);
        *position_ptr=moved;
        ptr->buffer[moved].value=value;
        ptr->buffer[moved].next=ptr->size;
        return TRUE;
    }else
        return FALSE;
}

```

```

Boolean ord_insert(struct list * ptr, float value)
/* Assume che la lista sia ordinata,
inserisce il nuovo elemento in ordine,
\`E analoga alla funzione di inserimento in coda salvo
l'aggiunta di una seconda condizione sul while
e uno swap nell'inserimento finale.
*/
{
    int moved;
    int * position_ptr;

    if(ptr->free!=ptr->size)
    {
        moved=ptr->free;
        ptr->free=((ptr->buffer)[ptr->free]).next;

        position_ptr=&ptr->first;
        while(*position_ptr!=ptr->size &&
              ptr->buffer[*position_ptr].value<value)
            position_ptr=&(ptr->buffer[*position_ptr].next);
        ptr->buffer[moved].value=value;
        ptr->buffer[moved].next=*position_ptr;
        *position_ptr=moved;
    }
}

```

```

        return TRUE;
} else
    return FALSE;
}

```

Le figure 2.4 e 2.5 illustrano il razionale dell'implementazione delle operazioni di inizializzazione e inserimento in testa. In generale, non esistono particolari criticità, se non nell'ordine con cui sono concatenati gli assegnamenti.

Un commento ulteriore è invece utile per il caso delle funzioni `suf_insert()` e `ord_insert()` che realizzano le operazioni di inserimento in coda o in posizione ordinata. Nel corpo della funzione viene definito un puntatore `position_ptr` che attraverso il ciclo `while` viene manipolato fino ad indirizzare il record sul quale deve essere operato l'inserimento. Se la lista è inizialmente vuota, l'indice del record su cui operare l'inserimento è quello contenuto nel campo `first` della lista; se invece la lista è non vuota allora l'indice del record su cui operare l'inserimento è contenuto nel campo `next` di uno dei records che stanno nel buffer. L'uso del puntatore `position_ptr` permette di unificare i due casi (vedi Fig.2.6).

Circa le due funzioni `suf_insert()` e `ord_insert()` è possibile anche osservare come le due siano sostanzialmente identiche salvo che nel caso di `ord_insert()` la guardia del ciclo `while` pone una condizione aggiuntiva che può terminare anticipatamente l'arresto nell'avanzamento del puntatore `position_ptr`.

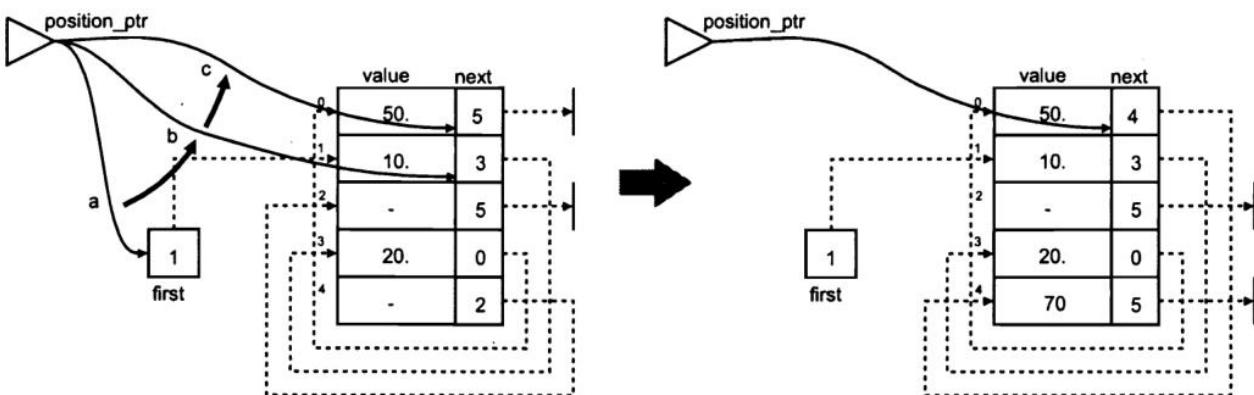


Figura 2.6: Nella operazione di inserimento in coda, la scansione della lista usa un puntatore che inizialmente contiene l'indirizzo dell'indice `first` (**a**); il puntatore viene avanzato (**b** e **c**) fino a che contiene l'indirizzo del campo `next` dell'ultimo record della lista (**c**); su quell'elemento viene effettuato l'inserimento.

La lista in forma collegata mediante indici mantiene il limite del dimensionamento statico del buffer che non permette di adattare la memoria allocata alla dimensione effettiva della sequenza rappresentata nel corso della computazione. Questo

che è un limite, è però anche il maggiore fattore di vantaggio rispetto alla implementazione collegata con puntatori: in effetti, il dimensionamento statico evita il bisogno di allocare e rilasciare memoria ad ogni operazione di inserimento e cancellazione, riducendo l'interazione con il sistema operativo e quindi aumentando sostanzialmente l'efficienza.

Come maggiore svantaggio rispetto alla forma sequenziale, si osservi che la rappresentazione collegata non permette l'accesso diretto agli elementi della lista. Nel caso limite, l'accesso all'ultimo elemento richiede che l'intera lista venga scandita con un numero di operazioni proporzionale al numero di elementi memorizzati.

2.1.3 Rappresentazione collegata con puntatori

Nella rappresentazione collegata con puntatori i valori sono memorizzati su elementi di memoria allocati separatamente in posizioni indipendenti. Ciascun valore è associato a un indirizzo che identifica la locazione di memoria nella quale si trova il valore successivo. Un indirizzo nullo codifica il termine della lista (vedi Fig.2.7).

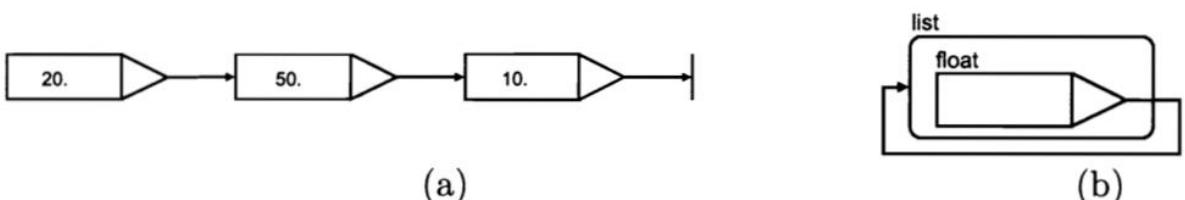


Figura 2.7: a) La rappresentazione della lista {10., 20., 50.} in forma collegata con puntatori. b) la relazione tra i tipi: una lista è una coppia composta di un valore (un **float** nell'esempio) e di un puntatore ad una lista.

```

struct list{
    float value;
    struct list * next_ptr;
};

void init(struct list ** ptrptr)
{
    *ptrptr=NULL;
}

void visit(struct list * ptr)
{
    while(ptr!=NULL)
    {
        printf("%f",ptr->value);
        ptr=ptr->next_ptr;
    }
}

```

```

Boolean search(struct list * ptr, float value, struct list ** ptrptr)
/* se esiste un elemento che contiene value, restituisce TRUE
   e iscrive in *ptrptr il puntatore all'elemento che contiene value.
   Altrimenti restituisce FALSE e non modifica *ptrptr
*/
{
    Boolean found;

    found=FALSE;
    while(ptr!=NULL&&found==FALSE)
    {   if(ptr->value==value)
        {   found=TRUE;
            *ptrptr=ptr;
        }
        else
            ptr=ptr->next_ptr;
    }
    return found;
}

void pre_insert(struct list ** ptrptr, float value)
/* inserimento in testa */
{
    struct list * tmp_ptr;

    tmp_ptr=*ptrptr;
    *ptrptr=(struct list *)malloc(sizeof(struct list));
    (*ptrptr)->value=value;
    (*ptrptr)->next_ptr=tmp_ptr;
}

Boolean consume_first(struct list ** ptrptr, float * value_ptr)
/* se la lista e' vuota restituisce FALSE;
   altrimenti restituisce TRUE, scrive in *value_ptr il
   valore nell'elemento in testa e lo cancella dalla lista */
{
    struct list * tmp_ptr;

    if(*ptrptr!=NULL)
    {   *value_ptr=(*ptrptr)->value;
        tmp_ptr=*ptrptr;
        *ptrptr=(*ptrptr)->next_ptr;
        free(tmp_ptr);
        return TRUE;
    }
    else
    {   return FALSE;
    }
}

```

```

void suf_insert(struct list ** ptrptr, float value)
/* inserimento in coda.
   Scorre il doppio puntatore fino a che questo punta un puntatore nullo,
   che per costruzione \`e il campo next_ptr dell'ultimo elemento della lista,
   ovvero il puntatore su cui applicare la malloc.
   Poi applica un inserimento in testa sulla sottolista puntata
   dal doppio puntatore */
{
    while(*ptrptr!=NULL)
        ptrptr=&((*ptrptr)->next_ptr);

    pre_insert(ptrptr,value);
}

void ord_insert(struct list ** ptrptr, float value)
/* inserimento in ordine.
   Simile all'inserimento in coda, salvo la condizione
   di arresto dell'avanzamento del doppio puntatore */
{
    while(*ptrptr!=NULL&& (*ptrptr)->value<value)
        ptrptr=&((*ptrptr)->next_ptr);

    pre_insert(ptrptr,value);
}

```

```

client( )
{
    float value;
    struct list * ptr;
    ...
    value=10.;
    pre_insert(&ptr,value);
    ...
}



---


pre_insert(struct list **ptrptr, float v)
{
    struct list * tmp_ptr;
    tmp_ptr=*ptrptr; // (1)
    *ptrptr=malloc(...); // (2)
    (*ptrptr)->value=v;
    (*ptrptr)->next_ptr=tmp_ptr; // (3)
}

```

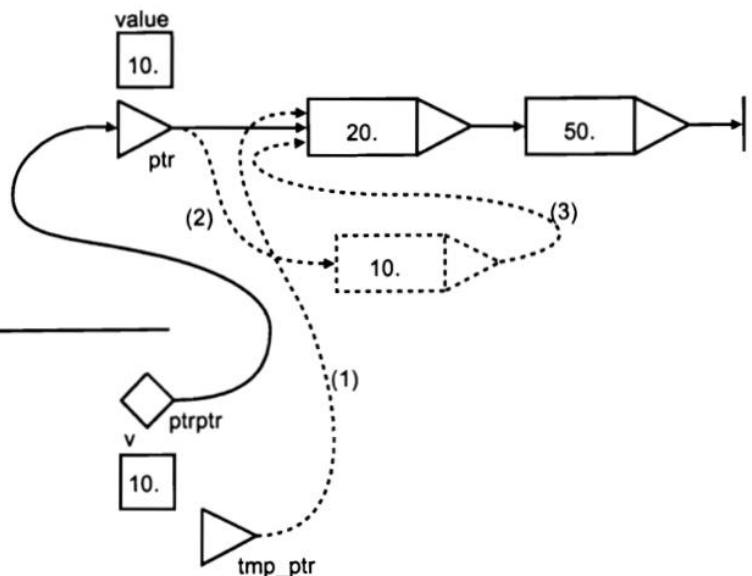


Figura 2.8: L'inserimento prefisso deve modificare il puntatore di testa della lista. Per questo la funzione `pre.insert()` deve ricevere l'indirizzo del puntatore stesso.

È significativo osservare l'uso di un doppio puntatore (ovvero di un puntatore che contiene l'indirizzo del puntatore ad una variabile) nelle funzioni di inserimento e nella inizializzazione. Consideriamo ad esempio il caso di `pre_insert()`. Questa deve modificare l'indirizzo del primo elemento della lista e quindi deve conoscere l'indirizzo del puntatore che contiene l'indirizzo del primo elemento della lista. Il modo in cui questo avviene è illustrato in Fig.2.8.

Vale la pena di soffermarsi a osservare cosa avverrebbe se anziché l'indirizzo del puntatore al primo elemento della lista venisse passato l'indirizzo del primo elemento della lista:

```
void client(void)
{ struct list * list_ptr;
  ...
  WRONG_pre_insert(list_ptr,5.);
}

void WRONG_pre_insert(struct list * ptr, float value)
/* versione non funzionante di un inserimento in testa */
{ struct list * tmp_ptr;
  tmp_ptr=ptr;
  ptr=(struct list *)malloc(sizeof(struct list));
  ptr->value=value;
  ptr->next_ptr=tmp_ptr;
}
```

```
client( )
{
  float value;
  struct list * ptr;
  ...
  value=10.;
  pre_insert(ptr,value);
  ...
}
```

```
WRONG_preinsert(struct list *ptr, float v)
{
  struct list * tmp_ptr;
  tmp_ptr=ptr;
  ptr=malloc(...);
  ptr->value=v;
  ptr->next_ptr=tmp_ptr;
}
```

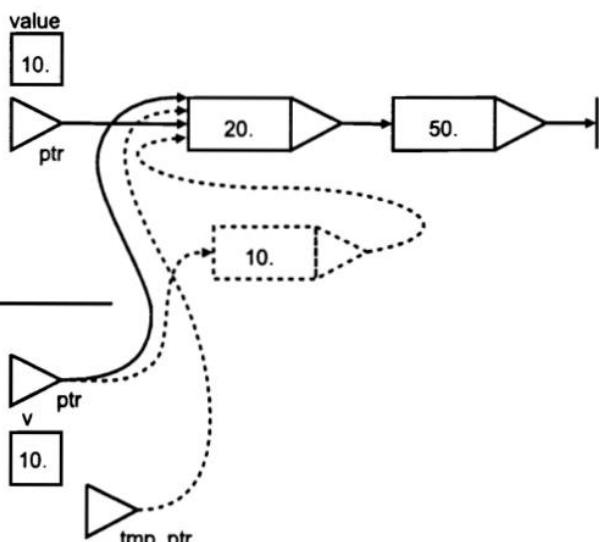


Figura 2.9: La funzione `WRONG.pre.insert` alloca un elemento della lista su una copia del puntatore di testa della lista. L'elemento è correttamente concatenato ma non è visibile nello spazio degli indirizzi della funzione chiamante.

L'effetto prodotto è illustrato in Fig.2.9: la funzione chiamante dispone di un puntatore alla testa della lista e ne passa una copia a `WRONG_pre_insert()`; questa crea un nuovo elemento della lista, lo antepone correttamente a quello che era il primo elemento della lista; ma non è in grado di modificare il puntatore `list_ptr` nello spazio delle variabili del chiamante. Dunque quando il controllo torna al `client()` questo non osserva alcuna modifica: la lista contiene ancora gli elementi che aveva prima della chiamata `WRONG_pre_insert()`; da qualche parte in memoria, irraggiungibile ad alcun riferimento, esiste un elemento di lista che contiene il valore 5 e punta la testa della lista.

È più sottile il caso dell'inserimento in coda o in posizione intermedia. Anche in questo caso la funzione riceve un doppio puntatore che contiene l'indirizzo del puntatore alla testa della lista.

Come è illustrato in Fig.2.10, se la lista è non vuota, il doppio puntatore viene avanzato lungo la lista puntando il campo `next_ptr` dei successivi elementi della lista fino a puntare il primo puntatore che contiene il valore `NULL`. Su questo puntatore viene effettuata la `malloc()` e viene attestato il nuovo elemento. Se la lista è non vuota, il puntatore modificato è il campo `next_ptr` di un elemento della lista.

Se però la lista è vuota o se comunque l'inserimento deve avvenire in testa per rispettare l'ordine, il puntatore da modificare è ancora quello che contiene l'indirizzo del primo elemento, come nel caso di un inserimento in testa. Per effettuare l'operazione, la funzione ha quindi bisogno di conoscere l'indirizzo del puntatore alla testa della lista.

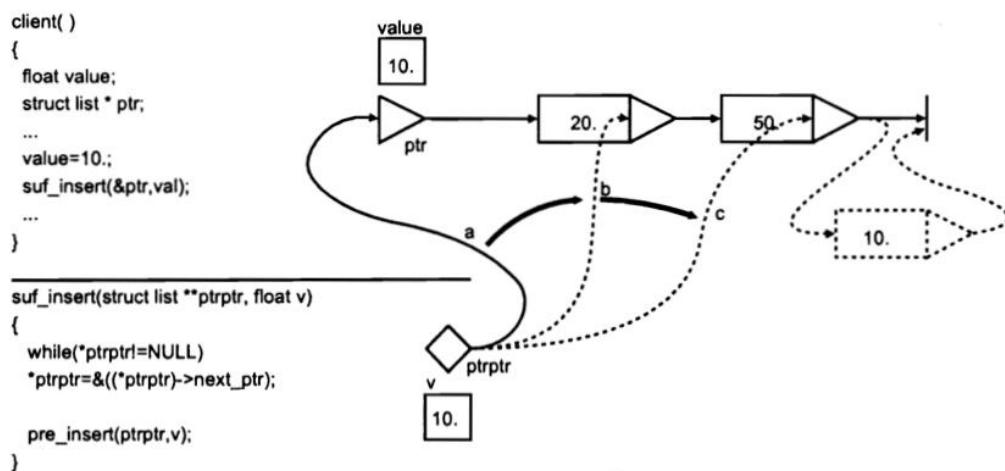


Figura 2.10: L'inserimento suffisso può dovere modificare il puntatore di testa della lista (nel caso in cui la lista sia vuota). Per questo la funzione `suf_insert()` deve ricevere l'indirizzo del puntatore stesso.

In generale non esiste modo di evitare il doppio puntatore a meno di fare ricorso al valore di ritorno della funzione. Questo conduce ad una implementazione che lascia al chiamante la responsabilità di aggiornare il puntatore di testa della lista:

```
struct list * pre_insert(struct list * ptr, float value)
/* inserimento in testa con puntatore semplice e valore di ritorno */
{ struct list * tmp_ptr;

    tmp_ptr=ptr;
    tmp_ptr=(struct list *)malloc(sizeof(struct list));
    tmp_ptr->value=value;
    tmp_ptr->next_ptr=ptr;

    return tmp_ptr;
}

void client(void)
{
    struct list * ptr;
    float value;
    ...
    ptr=init(ptr);
    ...
    ptr=pre_insert(ptr,value);
}
```

In generale, lasciare al chiamante la responsabilità di aggiornamento sul valore di testa della lista aumenta l'accoppiamento tra funzioni rendendone più difficile il riuso e la manutenzione separata. Esistono peraltro casi nei quali la possibilità per il chiamante di osservare il valore dell'indirizzo su cui è stato operato l'inserimento permette soluzioni efficienti per particolari problemi, tipicamente quando la stessa funzione viene invocata ripetutamente (o, come si dice, "a caldo").

Un esempio semplice ma significativo è il caso di una funzione `c` che riceve una lista collegata con puntatori e ne crea una copia (i.e. una lista che contiene gli stessi valori nello stesso ordine). La soluzione elementare consiste nello scandire la lista sorgente e inserire ciascun elemento visitato nella lista destinazione; per mantenere l'ordine degli elementi gli inserimenti devono essere effettuati in coda:

```
void clone(struct list * src_ptr, struct list ** dest_ptrptr)
{
    init(dest_ptrptr);

    while(src_ptr!=NULL)
    {
        suf_insert(dest_ptrptr,src_ptr->value);
        src_ptr=src_ptr->next_ptr;
    }
}
```

Il limite di questa soluzione è che essa comporta un *costo quadratico* rispetto al numero di elementi contenuti nella lista. Infatti, quando viene inserito l'elemento di ordine $k + 1$, la lista destinazione ha dimensione k e la funzione `suf_insert()` visita k elementi per arrivare a selezionare l'elemento di coda su cui effettuare l'inserimento. Così facendo, il numero complessivo di elementi visitati nel corso della operazione di copia dell'intera lista è pari a:

$$\Gamma_{clone}(N) = \sum_{k=0}^{N-1} c \cdot k = c \cdot \frac{N(N - 1)}{2} \quad (2.1)$$

Per ridurre la complessità al tempo lineare è sufficiente che la funzione `clone()` mantenga un puntatore all'ultimo puntatore della lista clonata per passarlo alla funzione `suf_insert()` come origine dalla quale avviare la scansione:

```
void clone2(struct list * src_ptr, struct list ** dest_ptrptr)
{
    init(dest_ptrptr);

    while(src_ptr!=NULL)
    {
        suf_insert(dest_ptrptr,src_ptr->value);
        dest_ptrptr=&((*dest_ptrptr)->next_ptr);
        src_ptr=src_ptr->next_ptr;
    }
}
```

Questo aumenta fortemente l'accoppiamento tra `clone2()` e `suf_insert()`, ma ha il vantaggio di evitare l'esecuzione del corpo del ciclo `while` nella funzione `suf_insert()` cosicché il numero di elementi visitati nel corso della intera operazione di copia della intera lista torna ad essere lineare rispetto al numero di elementi nella lista:

$$\Gamma_{clone2}(N) = \sum_{k=0}^{N-1} c = N \cdot c \quad (2.2)$$

2.1.4 Iterazione e ricorsione

La lista in forma collegata con puntatori offre una naturale occasione per discutere il meccanismo della ricorsione. In generale è ricorsiva una definizione che fa riferimento a sé stessa, in modo diretto o per il tramite di ulteriori definizioni intermedie. Questo può avvenire nella definizione della struttura di un dato o nella definizione di una funzione.

Ricorsione nei dati

La definizione della struttura `struct list` che rappresenta un elemento di una lista è ricorsiva perché include al suo interno il campo `next_ptr` che è un puntatore ad una struttura che è ancora del tipo `struct list`:

```
struct list{
    float value;
    struct list * next_ptr;
};
```

Questo è del resto un concetto già ampiamente esplorato nella definizione delle categorie sintattiche del linguaggio c e nella definizione della loro semantica.

Come unica annotazione aggiuntiva, osserviamo che nell'uso della ricorsione esistono alcune limitazioni, ragionevole conseguenza delle esigenze del compilatore: nel momento in cui il compilatore sviluppa la definizione di una struttura deve essere capace di costituire la mappa dei campi che la compongono, il che richiede che siano note le dimensioni e i formati di ciascuno dei campi che compaiono nella definizione.

Non è allora possibile includere nella definizione di una struttura un campo che sia esso stesso una struttura di formato non ancora definito. Come caso particolare, non è neppure legale il caso della definizione di una struttura che contiene un campo uguale a sé stesso, il che tra l'altro risulterebbe in una allocazione di spazio non limitata:

```
struct impossible_list{
    /* definizione illegale */
    float value;
    struct impossible_list;
}
```

La definizione di una struttura può invece contenere il puntatore ad una struttura non ancora definita (i.e. la cui definizione compare nel seguito del codice), visto che comunque la rappresentazione del puntatore ha la dimensione e il formato di un generico indirizzo, indipendente dal formato del tipo del particolare dato puntato. Questa è ad esempio il caso dell'indirizzo `next_ptr` che compare in modo ricorsivo all'interno della definizione di `struct list`.

Ricorsione nelle funzioni

La definizione di una funzione `f()` è ricorsiva quando essa delega parte della sua operazione ad una ulteriore istanza di `f()` stessa. Questo può avvenire in modo diretto, quando il corpo di `f()` contiene un riferimento a `f()` stessa, o in modo indiretto, quando il corpo di `f()` contiene un riferimento a una funzione che in

modo diretto o indiretto fa riferimento a `f()` (si noti che la definizione del concetto di ricorsione è essa stessa ricorsiva).

La ricorsione si applica in modo naturale nell'esecuzione di una operazione applicata a un insieme di dati, come potrebbe essere quello di una operazione su una lista. In questo caso, una forma ricorsiva si ottiene in modo naturale attribuendo alla funzione la responsabilità di trattare un singolo dato e delegando ad una sua ulteriore istanza il trattamento di tutti gli altri. Così facendo, le successive istanze nidificate ricevono la responsabilità di trattare un insieme di dati di dimensione strettamente decrescente fino a ridurre il problema al trattamento di un unico dato. Una condizione di guardia arresta la nidificazione delle chiamate quando l'insieme dei dati da trattare è esaurito.

Il concetto è esemplificato in modo elementare nel caso della operazione di visita:

```
void visit_r(struct list * ptr)
{
    if(ptr!=NULL)
    {
        printf("%f",ptr->value);
        visit_r(ptr->next_ptr);
    }
}
```

dovendo stampare una sequenza di valori, `visit_r()` ne stampa uno e delega ad una ulteriore istanza di `visit_r()` stessa di stampare gli elementi a partire dal successivo di quello già stampato; la guardia dell'`if` termina la nidificazione quando l'operazione di visita viene invocata su una lista vuota.

Il medesimo paradigma conduce alle seguenti implementazioni ricorsive per le altre operazioni sulla lista, fatta eccezione per l'inizializzazione e l'inserimento in testa che trattano un singolo dato e non richiedono quindi iterazione né ricorsione:

```
Boolean search_r(struct list * ptr, float value, struct list ** ptrptr)
/* ricerca in forma ricorsiva */
{
    if(ptr!=NULL)
    {
        if(ptr->value==value)
        {
            *ptrptr=ptr;
            return TRUE;
        }
        else
            return search_r(ptr=ptr->next_ptr, value, ptrptr);
    }
    else
        return found;
}

void suf_insert_r(struct list ** ptr, float value)
```

```

/* inserimento in coda in forma ricorsiva */
{
    if(*ptrptr!=NULL)
        suf_insert_r(ptrptr=&((*ptrptr)->next_ptr,value));
    else
        pre_insert(ptrptr,value);
}

void ord_insert_r(struct list ** ptr, float value)
/* inserimento in ordine in forma ricorsiva. */
{
    if(*ptrptr!=NULL&& (*ptrptr)->value<value)
        ord_insert_r(&((*ptrptr)->next_ptr),value);
    else
        pre_insert(ptrptr,value);
}

```

Affinità e divergenze tra gli schemi ricorsivi e iterativi

È interessante soffermarsi ad osservare le affinità che esistono nelle implementazioni in forma ricorsiva e iterativa di una stessa operazione. Tali affinità sono in larga parte sistematiche, reggendosi su un sostanziale dualismo nel modo con cui iterazione e ricorsione risolvono il comune problema di ripetere l'esecuzione di uno stesso frammento di codice su una molteplicità di variabili.

Focalizziamo ancora l'analisi sul caso della funzione di visita.

- Nella forma iterativa, l'operazione elementare `printf()` viene applicata a una molteplicità di variabili collocandola nel corpo di un ciclo `while`; dopo l'esecuzione dell'operazione elementare, l'avanzamento del puntatore `ptr=ptr->next_ptr` aggiorna l'indirizzo della variabile che sarà trattata nella prossima iterazione.

Nella forma ricorsiva, l'istruzione che realizza l'operazione elementare è esattamente la stessa ma in questo caso l'aggiornamento sul dato da trattare è ottenuto modificando il parametro con cui viene attivata la funzione `visit_r()` stessa: nella posizione del parametro formale di ingresso `ptr` viene ora passata l'espressione `ptr->next_ptr`. Questo riproduce lungo la direzione di nidificazione delle successive istanze della funzione `visit_r()` lo stesso effetto di aggiornamento del dato da trattare prodotto nella forma iterativa con l'assegnamento `ptr=ptr->next_ptr`.

Al di là di considerazioni circa la naturalezza e l'eleganza delle diverse soluzioni, è importante rimarcare come lo schema ricorsivo incorra in un maggiore costo in termini di spazio di memoria e tempo di esecuzione, dovuto al diverso carico sullo stack di sistema. Nello schema ricorsivo, l'esecuzione

della visita comporta la nidificazione di un numero di istanze della funzione `vist_r()` pari alla lunghezza della lista; per ogni istanza, sullo stack viene copiato l'indirizzo di ritorno della funzione e il valore dei parametri attuali; se ce ne fossero, sarebbero allocate anche le eventuali variabili locali dichiarate nel corpo della funzione. Questo fattore di costo aggiuntivo rispetto allo schema iterativo è generale, ed è la buona ragione per cui soluzioni ricorsive non sono adatte a implementare operazioni che incidono significativamente sulla prestazione di un sistema.

- Gli schemi ricorsivi hanno talvolta una maggiore capacità di adattarsi alla struttura dei dati. Questo è in ultimo la ragione per cui in molti casi, pur a costo di una minore efficienza di esecuzione, uno schema ricorsivo può risultare migliore.

Un esempio che illustra bene il concetto è il problema della visita di una lista in ordine inverso. In sostanza, assegnata una lista {1, 2, 5} si vuole vedere comparire a video la sequenza {5, 2, 1}. In uno schema ricorsivo, l'effetto di invertire l'ordine di visita degli elementi si ottiene semplicemente postponendo l'operazione sul dato corrente (nell'esempio l'istruzione `printf()`) alla distribuzione del controllo mediante la chiamata ricorsiva:

```
void visit_r_backward(struct list * ptr)
{
    if(ptr!=NULL)
    {
        visit_r(ptr->next_ptr);
        printf("%f",ptr->value);
    }
}
```

Ottenerlo lo stesso effetto in uno schema iterativo è più complesso. In questo caso, la soluzione migliore è quella di copiare i valori su una lista di appoggio che inverte la relazione di successione e poi scandire la lista di appoggio. La lista invertita si genera in modo naturale scandendo la lista originale e inserendo in testa alla lista di appoggio:

```
void visit_backward(struct list * ptr)
{ struct list * tmp_ptr;

    init(&tmp_ptr);
    while(ptr!=NULL)
    {
        pre_insert(&tmp_ptr,ptr->value);
        ptr=ptr->next_ptr;
    }
    visit(tmp_ptr);
}
```

Poiché gli inserimenti sono effettuati in testa, la lista di appoggio viene creata in tempo lineare con un numero di operazioni proporzionale alla dimensione della lista. Dunque non viene modificato l'ordine di grandezza della complessità della visita che è comunque di per sé una operazione id complessità lineare.

Rispetto allo schema ricorsivo emerge il fatto che la lista di appoggio richiede uno spazio di memoria proporzionale alla dimensione della lista, il che apparentemente non avviene nell'implementazione ricorsiva.

Ad una analisi attenta del carico sullo stack di sistema, si osserva però che anche la soluzione ricorsiva incorre nella stessa occupazione di memoria: in quel caso gli elementi visitati e non ancora stampati a video sono memorizzati come parametri formali sullo stack di sistema; quando la ricorsione raggiunge la sua massima nidificazione, sullo stack sono copiati gli indirizzi di tutti i nodi della lista.

Rimane il fatto che l'implementazione ricorsiva ha il pregio di rendere trasparente al programmatore la memorizzazione delle variabili intermedie, godendo anche in qualche misura di una maggiore efficienza nella copia di variabili sullo stack di sistema piuttosto che in aree di memoria allocate dal programma mediante l'istruzione `malloc()` in `pre_insert()`.

- Nella forma iterativa della operazione di visita, il controllo sulla terminazione della ripetizione è effettuato in testa prima del corpo del `while`. Allo stesso modo, nella forma ricorsiva, il controllo di terminazione è effettuato da un `if` in testa al corpo della funzione.

Nell'approccio iterativo la terminazione della ripetizione può essere controllata in uscita, usando un `do while` al posto del `while`. Questo implica che all'inizio di una iterazione è garantito che l'insieme dei dati da trattare è non vuoto, e di converso la struttura di controllo permette la ripetizione solo se dopo il trattamento di un dato è garantito che ce ne siano ancora altri. Ovviamente è necessario aggiungere un controllo iniziale, eseguito una sola volta, per verificare se l'insieme iniziale da trattare è non vuoto. La soluzione non è elegante ma può avere un suo razionale:

```
void visit_p(struct list * ptr)
{
    if(ptr!=NULL)
    {   do
        {   printf("%f",ptr->value);
            ptr=ptr->next_ptr
        }while(ptr!=NULL)
    }
}
```

Passando ad uno schema ricorsivo, il controllo in uscita si traduce nell'attribuire al chiamante la responsabilità del controllo sulla legalità del dato da trattare: il corpo della funzione assume che il suo chiamante abbia già provveduto a verificare che l'insieme dei dati da trattare è non vuoto; di converso, prima di propagare il controllo con una chiamata ricorsiva, la funzione controlla che l'insieme dei dati che rimangono da trattare sia non vuoto. Questo risulta nello schema:

```
void visit_rp(struct list * ptr)
/* assume che sia ptr!=NULL */
{
    printf("%f",ptr->value);
    if(ptr->next_ptr!=NULL)
        visit_rp(ptr->next_ptr);
}
```

In questo caso, la responsabilità di verificare che la prima attivazione della funzione sia legale deve essere collocata nella funzione esterna che usa `visit_rp()`, in modo che essa sia eseguita una sola volta:

```
void client(void)
{
    struct list * ptr;
    ...

    if(ptr!=NULL)
        visit_rp(ptr);
    ...
}
```

- Nell'approccio ricorsivo, l'uso di una funzione di facciata che assume la responsabilità di operazioni eseguite una sola volta è spesso il modo di realizzare operazioni di inizializzazione e chiusura che in uno schema iterativo sarebbero eseguite prima e dopo l'istruzione di iterazione. Talvolta il problema può essere evitato attribuendo la responsabilità delle operazioni eseguite una sola volta all'istanza della funzione che riconosce di operare su un insieme vuoto di dati.

Illustriamo il concetto discutendo l'operazione di somma dei valori degli elementi di una lista. Si tratta in sostanza di una estensione della visita nella quale i valori visitati sono sommati su una variabile di accumulazione. In forma iterativa, questo è realizzato nello schema:

```
void sum(struct list * ptr, float * sum_ptr)
```

```

{
    *sum_ptr=0;
    while(ptr!=NULL)
    {
        (*sum_ptr)+=ptr->value;
    }
}

```

Nella forma ricorsiva equivalente, la responsabilità dell'inizializzazione della somma dovrebbe essere delegata a una funzione di facciata esterna:

```

void client(void)
{
    struct list * ptr;
    int sum;
    ...
    sum=0
    sum_r(ptr, &sum);
    ...
}

void sum_r(struct list * ptr, float * sum_ptr)
{
    if(ptr!=NULL)
    {
        (*sum_ptr)+=ptr->value;
        sum_r(ptr->next_ptr,sum_ptr);
    }
}

```

L'uso della facciata esterna può essere evitato eseguendo la visita in forma posticipata e attribuendo la responsabilità dell'inizializzazione alla prima istanza della funzione che opera su una lista nulla:

```

void sum_r_backward(struct list * ptr, float * sum_ptr)
{
    if(ptr!=NULL)
    {
        sum_r(ptr->next_ptr,sum_ptr);
        (*sum_ptr)+=ptr->value;
    }else
    {
        *sum_ptr=0;
    }
}

```

Si osservi che laddove siano contemporaneamente necessarie operazioni di inizializzazione e di chiusura, lo schema dell'inversione dell'ordine di visita non è più sufficiente e diventa invece necessario l'uso di una facciata che attribuisce al chiamante una delle due operazioni o entrambe.

- Nelle implementazioni in forma ricorsiva, l'uso del valore di ritorno delle funzioni permette spesso di ottenere soluzioni di maggiore semplicità e di ridurre il numero di variabili locali e parametri caricati sullo stack di sistema.

Illustriamo il concetto facendo ancora riferimento alla operazione di somma dei valori degli elementi di una lista, già implementata precedentemente nella funzione `visit_r_backward()`. Utilizzando il valore di ritorno al posto del secondo parametro, l'operazione può essere implementata nella forma:

```
int sum_r_return(struct list * ptr)
{
    if(ptr!=NULL)
        { return ptr->value+sum_r_return(ptr->next_ptr,sum_ptr);
    }else
        { return 0;
    }
}
```

che risolve in modo naturale il problema della inizializzazione della variabile di accumulazione della somma e riduce il numero di variabili allocate sullo stack.

È utile soffermarsi sulla diversa natura delle due istruzioni di `return`. Mentre l'istruzione `return 0;` determina un risultato terminale, l'istruzione `return ptr->value+sum_r_return(ptr->next_ptr,sum_ptr);` propaga il valore restituito da altre istanze della funzione aggiungendoci qualcosa.

Il concetto è illustrato in modo più elaborato nel caso di una operazione di ricerca su una lista ordinata. In questo caso la terminazione può avvenire perché viene identificato un elemento che contiene il valore cercato oppure perché la visita esaurisce la lista o raggiunge un valore maggiore di quello cercato:

```
Boolean search_r_return(struct list * ptr, float target)
{
    if(ptr!=NULL && ptr->value<=target)
        { if(ptr->value==target)
            return true;
        else
            return search_r_return(ptr->next_ptr,target);
    }else
        { return false;
    }
}
```

2.2 Alberi

L'albero è un insieme di elementi, detti *nodi*, sui quali è definita una relazione di discendenza che gode di due proprietà: esiste unico un nodo, detto *radice*, che non ha predecessori; qualsiasi altro nodo ha un unico predecessore. I nodi che non hanno successori sono detti *foglie*; un nodo che non è né la radice né una foglia si dice *intermedio*. Fig.2.11 illustra il concetto: l'elemento x è la radice e ha due successori y e z ; z ha tre successori t , u e v ; i nodi y , t , u e v sono foglie. Si osservi che un albero in cui ciascun nodo ha al massimo un successore degenera nella forma di una lista.

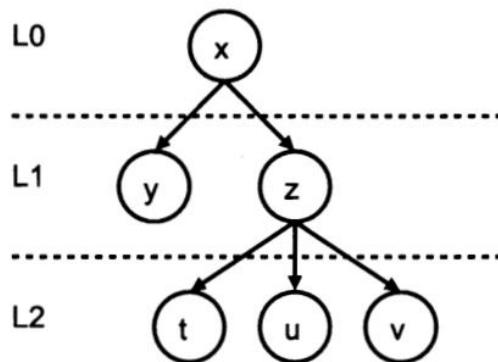


Figura 2.11: Un albero: x è la radice, z è un nodo intermedio, y , t , u , v sono foglie; z realizza il massimo grado di uscita, pari a 3; l'albero è ripartito su tre livelli ed ha quindi ha profondità 2.

Il concetto può essere formalmente catturato definendo l'albero come una coppia $tree = \langle V, E \rangle$ dove V è un insieme di valori di un qualche tipo, ed $E \subseteq V \times V$ è una relazione su V , ovvero un insieme di coppie $e = \langle v_{parent}, v_{child} \rangle$ con *parent* e *child* elementi di V . Gli elementi di V sono detti vertici e quelli di E archi; dati due vertici v_{parent} e v_{child} si dice che v_{parent} è predecessore di v_{child} e viceversa che v_{child} è successore di v_{parent} se E contiene la coppia $\langle v_{parent}, v_{child} \rangle$. La relazione E gode delle seguenti proprietà: esiste unico un vertice (la radice) che non è successore di alcun altro vertice (i.e. $\exists_1 root \in V . \forall (v_1, v_2) \in E v_2 \neq root$); qualsiasi altro vertice ha uno e un solo predecessore (i.e. $\forall v, v_{p1}, v_{p2} \in V, \text{ se } \langle v_{p1}, v \rangle \in E \text{ e } \langle v_{p2}, v \rangle \in E \text{ allora } v_{p1} = v_{p2}$).

Nella caratterizzazione di un albero hanno rilevanza il *grado di uscita* e la *profondità* dei nodi. Il grado di uscita di un nodo è il numero dei suoi successori diretti. La profondità definisce la distanza di un nodo dalla radice ed è convenientemente definita in modo ricorsivo: la radice ha profondità 0; un generico nodo diverso dalla radice ha profondità pari a quella del suo predecessore aumentata di 1. Per estensione, la profondità di un albero è il massimo delle profondità dei

diversi nodi. L'insieme dei nodi che si trovano ad una stessa profondità forma un *livello*.

Su un albero si eseguono sostanzialmente le stesse operazioni che si applicano anche ad una lista, con un paio di differenze: su un albero non esiste un unico nodo terminale ma esiste invece una molteplicità di foglie, per cui l'operazione di inserimento in coda deve essere qualificata con un qualche criterio per selezionare quale tra le diverse foglie è il target dell'operazione; l'inserimento sulla radice o su nodi intermedi tendono a degenerare la struttura dell'albero verso una forma sequenziale, per cui gli inserimenti sono tipicamente eseguiti sulle foglie.

2.2.1 Alberi binari di ricerca

Un albero binario è un albero sul quale ciascun nodo ha grado di uscita minore o uguale a 2. Un albero binario si dice *di ricerca* quando il valore codificato su ciascun nodo è maggiore o uguale del valore codificato sul figlio sinistro del nodo stesso e minore secco del valore codificato sul figlio destro. Entrambi gli alberi in Fig.2.12 realizzano la condizione.

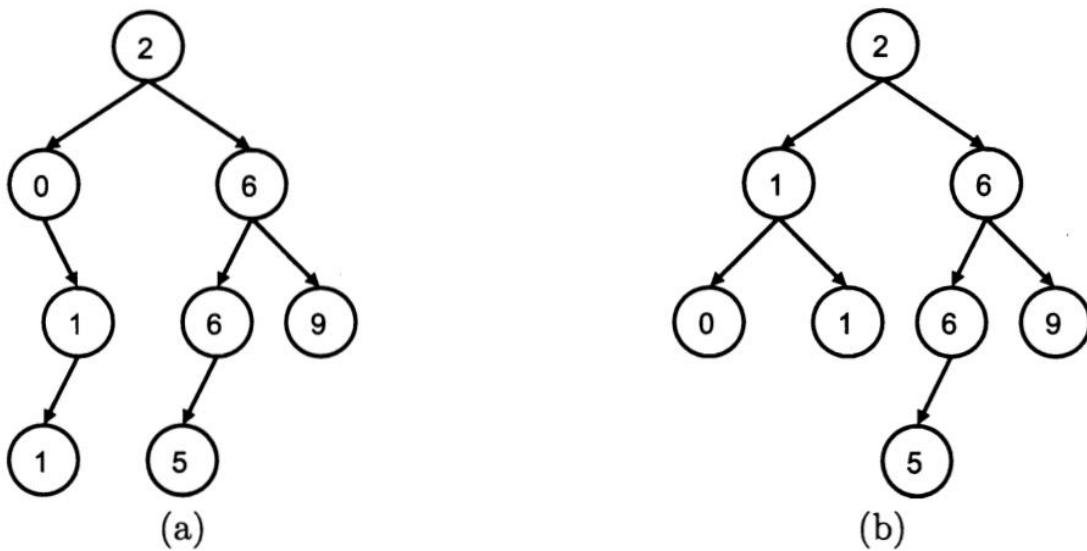


Figura 2.12: Due alberi binari di ricerca. L'albero (a) non è bilanciato, il (b) lo è.

Un albero binario si dice *bilanciato* quando ogni nodo che si trovi su un livello diverso dall'ultimo o dal penultimo ha due figli. L'albero in Fig.2.12a non soddisfa la condizione perché il nodo che contiene il valore 0 ha a un sottoalbero destro di profondità 2 e un sottoalbero sinistro di profondità 0. La condizione è invece realizzata nell'albero di Fig.2.12b. Ragionando induttivamente sulla dimensione dei sottoalberi è facile verificare che un albero binario di profondità D è bilanciato se e solo esso contiene tutti i nodi che possono essere ospitati sui livelli fino a $D - 1$ incluso. Da questo deriva un limite sulla profondità massima in funzione

del numero dei nodi. Infatti, su un albero bilanciato di profondità D , la condizione di completezza dei primi $D - 1$ livelli e la esistenza di un nodo almeno sul livello di profondità D implica che il numero N dei nodi soddisfi la disequazione:

$$N \geq 1 + \sum_{d=0}^{D-1} 2^d = 2^D \quad (2.3)$$

da cui segue che la profondità di un albero binario bilanciato è limitata dall'alto dal logaritmo del numero dei nodi:

$$D \leq \ln_2(N) \quad (2.4)$$

L'albero binario può essere rappresentato in forma sequenziale, o in forma collegata a seconda che la relazione di discendenza tra i nodi sia codificata implicitamente attraverso la posizione occupata dai nodi in un vettore oppure se essa è rappresentata in modo esplicito con l'uso di indici o puntatori. Discutiamo nel seguito le tre diverse rappresentazioni evitando per quanto possibile di ripetere concetti già esposti nella trattazione delle liste.

2.2.2 Forma collegata con puntatori

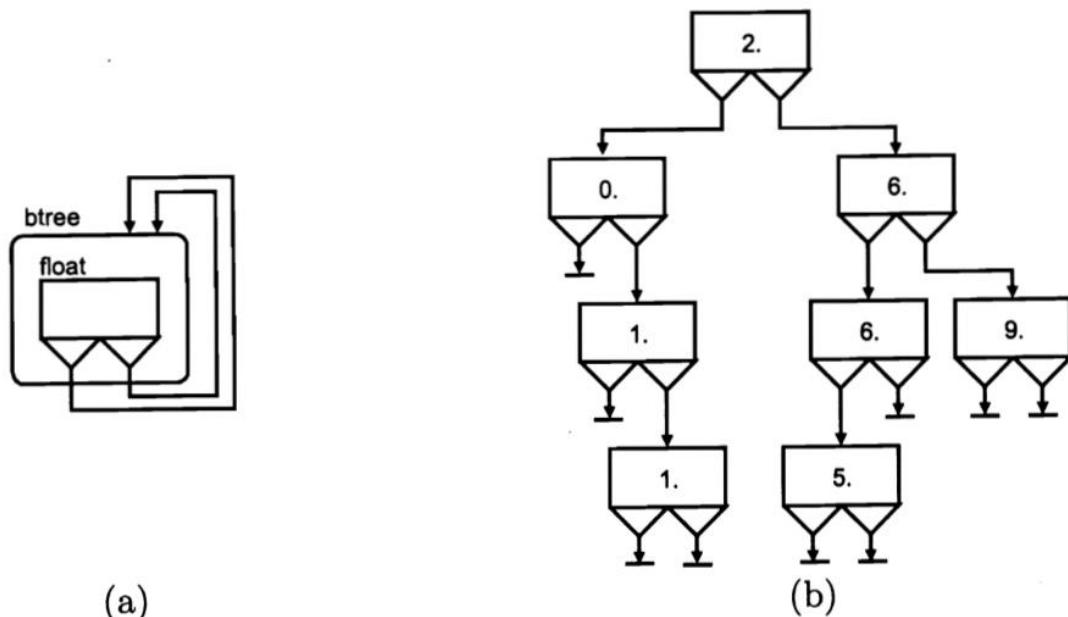


Figura 2.13: (a) La rappresentazione di un albero binario in forma collegata con puntatori: un nodo è composto di un valore (un float nell'esempio) e di due puntatori a due nodi. (b) La rappresentazione in forma collegata dell'albero binario di Fig.2.12a.

La rappresentazione dell'albero binario in forma collegata con puntatori estende in modo naturale quella già applicata nella rappresentazione di una lista: un nodo contiene un valore e due puntatori ai nodi figli; dove uno o entrambi i figli non esistono, il puntatore corrispondente ha il valore **NULL** (Fig.2.13)

```
struct btree{  
    float value;  
    struct btree * leftptr;  
    struct btree * rightptr;  
};
```

Visita in forma ricorsiva

L'implementazione della visita, almeno nella forma ricorsiva, è anch'essa sostanzialmente equivalente a quella già discussa per la lista:

```
void visit_r(struct btree * ptr)  
/* visita simmetrica */  
{  
    if(ptr!=NULL)  
    {  
        visit_l(ptr->left_ptr);  
        printf("%f",ptr->value);  
        visit_r(ptr->right_ptr);  
    }  
}
```

È interessante osservare l'ordine nella visita dell'albero: la radice viene trattata solo dopo avere trattato tutti i nodi del sottoalbero sinistro, e prima di trattare alcun nodo del sottoalbero destro. Così facendo, se l'albero è di ricerca, i valori sono stampati in ordine. Infatti, ragionando induttivamente sulla dimensione dei sottoalberi, è facile verificare che il valore di un qualsiasi nodo è stampato solo dopo che sono stati stampati tutti e soli i valori che gli sono minori o uguali e prima di stampare alcuno dei valori che gli sono maggiori.

Nel momento in cui viene visitato un generico nodo, sullo stack di sistema sono caricati i parametri di un numero di istanze della funzione di visita pari alla profondità del nodo stesso. Dunque il carico massimo è proporzionale alla profondità dell'albero

Si osservi che, come già discusso nella forma ricorsiva della visita di una lista, la posizione mutua delle tre istruzioni sotto la guardia dell'*if* impatta sull'ordine con cui sono visitati i nodi. L'implementazione riportata sopra si dice in forma *simmetrica*. Quando il trattamento del dato visitato è posto in prima posizione si ottiene una visita in *pre-ordine*, e quando è posto in terza posizione si dice in *post-ordine*. In particolare, è utile osservare che nella visita in post-ordine un nodo

viene visitato solo dopo che sono stati visitati tutti i nodi della sua discendenza, il che si dice essere una visita *depth-first* (in profondità). In qualsiasi delle tre forme di visita, il carico massimo sullo stack rimane comunque proporzionale alla profondità dell'albero.

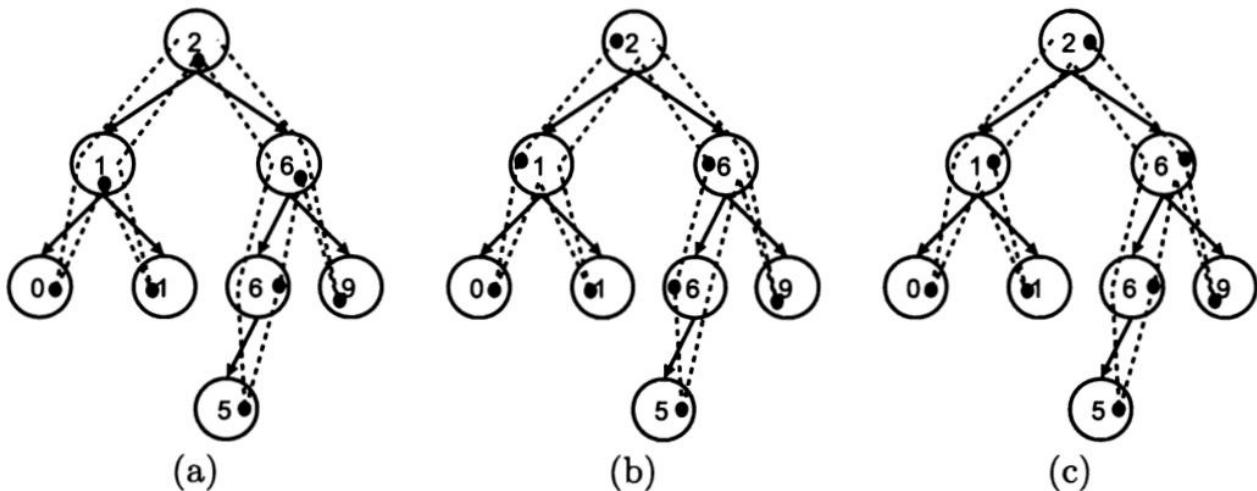


Figura 2.14: L'ordine di visita dei nodi dell'albero binario di Fig.2.12a nelle implementazioni ricorsive in forma: (a) simmetrica; (b) anticipata; (c) posticipata. La linea tratteggiata rappresenta il flusso del controllo, i punti marcati lungo la linea al momento al quale viene stampato il valore del nodo visitato.

Visita in forma iterativa

L'implementazione in forma iterativa della visita esemplifica concretamente il prezzo dello schema ricorsivo nell'adattarsi alla struttura dei dati. In effetti, su un albero, la visita di un nodo genera due prossimi nodi da visitare. Nello schema ricorsivo questo è risolto facilmente accompagnando la stampa del valore di ciascun nodo a due chiamate ricorsive che effettuano la visita sui sottoalberi che hanno radice nei due nodi figli. Nello schema iterativo questo non è possibile, essendo necessario avere sempre un unico prossimo dato da trattare.

Il problema può essere risolto, con un accorgimento di validità generale, realizzando una struttura dati che raccoglie i nodi raggiunti ma non ancora trattati, il che in ultimo riproduce il ruolo che nella forma ricorsiva è svolto dallo stack di sistema. Se assumiamo che `set` sia una struttura dati che rappresenta un insieme di indirizzi di nodi di un `btree` la quale implementa le operazioni di inserimento e consumo di un elemento (`add` e `get`), il test di non-vuoto (`is_empty`) e l'inizializzazione (`init`), la funzione `visit_r` può essere tradotta nella seguente forma iterativa:

```
void visit(struct btree * ptr)
{
```

```

struct set * tobevisited_ptr;
init(&tobevisited_ptr);

add(&tobevisited_ptr,ptr);
while(not_empty(tobevisited_ptr))
{
    this_node_ptr=get(&tobevisited_ptr);

    add(&tobevisited_ptr,this_node_ptr->left_ptr);
    printf("%f",this_node_ptr->value);
    add(&tobevisited_ptr,this_node_ptr->right_ptr);
}

```

L'insieme set può essere rappresentato in vario modo e gestito con diverse politiche di ingresso e uscita; nella soluzione più immediata viene usata una lista in forma collegata su cui si implementano inserimento e consumo con una inserzione e una cancellazione in testa. Per ogni nodo raggiunto, l'algoritmo stampa il valore del nodo stesso e poi aggiunge i puntatori dei figli (se esistono) alla lista dei nodi da stampare:

```

struct list
/* lista di indirizzi di btree */
{
    struct btree * ptr;
    struct list * next_ptr;
};

void visit(struct btree * ptr)
{
    struct list * tobevisited_ptr;

    init(&tobevisited_ptr);
    pre_insert(&tobevisited_ptr,ptr);
    while(tobevisited_ptr!=NULL)
    {
        this_node_ptr=consume_first(&tobevisited_ptr);
        printf("%f",this_node_ptr->value);
        if(this_node_ptr->left_ptr!=NULL)
            pre_insert(&tobevisited_ptr,this_node_ptr->left_ptr);
        if(this_node_ptr->right_ptr!=NULL)
            pre_insert(&tobevisited_ptr,this_node_ptr->right_ptr);
    }
}

```

Poiché gli inserimenti e le cancellazioni sulla lista sono effettuati entrambi in testa, la coda degli indirizzi da trattare è gestita con politica First In Last Out. Sulla base di questo possiamo venire ad alcune conclusioni sull'ordine con cui sono visitati i nodi dell'albero: un nodo è stampato prima di qualsiasi suo discendente; il figlio destro è stampato prima del figlio sinistro; il figlio sinistro viene stampato solo

dopo che è stata stampata tutta la discendenza del figlio destro. Per quest'ultima proprietà la sequenza di visita è ancora di tipo depth-first.

Il numero massimo di indirizzi memorizzati sulla lista è pari alla profondità dell'albero e si realizza quando la visita raggiunge il nodo più sinistro e di più basso livello: in quel momento sulla lista sono caricati gli indirizzi dei fratelli di tutti gli antenati del nodo stesso. Nel caso di un albero bilanciato questo è limitato da $\ln_2(N)$, dove N è il numero complessivo dei nodi nell'albero. Tuttavia, nel caso pessimo di un albero degenere nel quale ciascun livello contiene due soli nodi, il carico può diventare pari a $\frac{N}{2}$.

Se l'insieme dei nodi da trattare è gestito con politica First In First Out, l'albero viene visitato in ampiezza (in modo *breadth-first*): prima è visitata la radice, poi tutti i nodi sul primo livello, da sinistra a destra, poi tutti i nodi sul secondo livello da sinistra a destra, e così via fino a che la visita termina sul nodo più destro del livello più basso. L'effetto si può ottenere operando inserimenti e cancellazioni in posizioni opposte, ad esempio inserimenti in coda e cancellazioni in testa.

L'inserimento in coda deve essere implementato in modo raffinato per evitare che in ciascuna delle successive operazioni si abbia un costo proporzionale alla lunghezza della lista. Come già discusso nella trattazione del problema della copia di una lista, la condizione si può ottenere con una implementazione di `suf_insert` che restituisca l'indirizzo dell'elemento di lista inserito:

```
void visit_breadth_first(struct btree * ptr)
{
    struct list * tobevisited_ptr;
    struct list * tail_ptr

    init(&tobevisited_ptr);
    tail_ptr=suf_insert(tobevisited_ptr,ptr);
    while(tobevisited_ptr!=NULL)
    {
        this_node_ptr=consume_first(&tobevisited_ptr)
        printf("%f",this_node_ptr->value);
        if(this_node_ptr->left_ptr!=NULL)
            tail_ptr=suf_insert(tail_ptr,this_node_ptr->left_ptr);
        if(this_node_ptr->right_ptr!=NULL)
            tail_ptr=suf_insert(tail_ptr,this_node_ptr->right_ptr);
    }
}
```

Anche con questo accorgimento, la visita breadth-first ha comunque un limite inerente nel carico sulla lista di appoggio. Infatti, nel momento in cui la visita completa il penultimo livello, sulla lista sono memorizzati tutti i nodi dell'ultimo livello. Nel caso di un albero bilanciato questo può essere pari alla metà (più 1) di tutti i nodi presenti nell'intero albero: $2^D = 1 + \sum_{d=0}^{D-1} 2^d = \frac{N+1}{2}$.

Ricerca

Su un albero binario di ricerca è possibile eseguire una ricerca in modo efficiente, evitando di visitare esaustivamente tutti i nodi. Questa è in ultimo la più frequente motivazione per l'uso di un albero binario.

L'algoritmo di ricerca è convenientemente descritto in forma ricorsiva: se la radice contiene il valore target, la ricerca termina con successo; se il valore della radice è maggiore (minore) del target allora questo può trovarsi solo nel sottoalbero sinistro (destro) per cui l'esito della ricerca è quello della ricerca sul sottoalbero sinistro (destro); la ricerca fallisce appena viene raggiunto un sottoalbero vuoto:

```
Boolean search(struct btree * ptr, float target)
{
    if(ptr!=NULL)
    {
        if(ptr->value==target)
        {
            return TRUE;
        }
        else
        {
            if(value<ptr->value)
                return search(ptr->left_ptr, target);
            else
                return search(ptr->right_ptr, target);
        }
    }
    else
        return FALSE;
}
```

È utile osservare che (diversamente da quanto avviene per la operazione di visita) l'algoritmo procede lungo un percorso sequenziale perché ad ogni nodo visitato la ricerca deve essere propagata solo verso uno dei due sottoalberi. Questo permette di tradurre l'implementazione in forma iterativa secondo meccanismi già discussi per le liste:

```
Boolean rec_search(struct btree * ptr, float target)
{
    Boolean found;

    found=FALSE;
    while(ptr!=NULL)
    {
        if(ptr->value==target)
        {
            found=TRUE;
        }
        else
        {
            if(value<ptr->value)
                ptr=ptr->left_ptr;
            else
                ptr=ptr->right_ptr;
        }
    }
}
```

```

    }
}

return found;
}

```

È importante osservare che su ciascun livello dell'albero viene visitato al massimo un singolo nodo, per cui l'algoritmo termina dopo avere visitato al massimo un numero di nodi pari alla profondità dell'albero. Come caso particolare, se l'albero è bilanciato, il numero dei nodi visitati è minore o uguale a $\ln_2(N)$. Se l'albero degenera in forma lineare, la profondità può invece essere pari al numero N stesso dei nodi memorizzati.

Inserimento

L'inserimento in testa perde significato nel caso di un albero binario. È vero che l'inserimento in testa può essere eseguito in tempo costante, ma è chiaro che un albero costruito inserendo sempre in testa degenera verso una forma lineare con profondità pari al numero dei nodi memorizzati.

L'inserimento in coda a sua volta pone il problema di stabilire quale sia la foglia. Nel caso di un albero binario di ricerca il problema è risolto inserendo ogni nuovo valore nell'unica foglia che mantiene l'albero in ordine. Per questo l'algoritmo discende l'albero a partire dalla radice. Se l'albero è non vuoto l'inserimento è delegato ad una chiamata ricorsiva che opera sull'unico dei due sottoalberi abilitati a contenere il valore da inserire. Quando la ricorsione raggiunge una foglia viene finalmente effettuato l'inserimento:

```

void insert_inorder(struct btree ** ptrptr, value)
{
    if(*ptrptr!=NULL)
    {
        if(value<=(*ptrptr)->value)
            insert_inorder(&ptr->left_ptr, value);
        else
            insert_inorder(&ptr->right_ptr, value);
    }
    else
    {
        (*ptrptr)=(struct btree *)malloc(sizeof(struct btree));
        (*ptrptr)->value=value;
        (*ptrptr)->left_ptr=NULL;
        (*ptrptr)->right_ptr=NULL;
    }
}

```

Si osservi l'uso del doppio puntatore, che estende quanto già discusso in riferimento alla implementazione dell'inserimento in coda su una lista.

Si osservi anche che, come per la ricerca, anche l'inserimento esegue un percorso sequenziale lungo l'albero in quanto ciascun nodo propaga la visita verso uno solo dei suoi successori. Per questo il numero di nodi visitati è proporzionale alla profondità dell'albero che, nel caso di un albero bilanciato è pari al logaritmo in base 2 del numero di nodi contenuti nell'albero. Per la stessa ragione, l'algoritmo può essere tradotto in forma iterativa in modo diretto.

Cancellazione

L'operazione di cancellazione è diversa a seconda del numero di figli del nodo che deve essere cancellato.

Per cancellare un nodo foglia è sufficiente annullare il puntatore al nodo stesso da cancellare (e poi magari rilasciare la memoria allocata).

È semplice anche la cancellazione di un nodo che ha un unico successore: in tal caso sarà sufficiente redirigere il puntatore al nodo che deve essere cancellato in modo che esso vada a indirizzare l'unico figlio del nodo cancellato (trasformazione (ii) in Fig.2.15).

Per cancellare un nodo che ha due successori è possibile ricondursi al caso di un unico successore spostando uno dei due sottoalberi (ad esempio, nella trasformazione (i) di Fig.2.15 viene spostato il sottoalbero sinistro). Per mantenere la condizione di ordinamento propria di un albero di ricerca, il sottoalbero spostato deve essere riagganciato sull'estremo appropriato del sottoalbero che è rimasto nella sua posizione. Se il sottoalbero spostato è il sinistro, la sua radice deve diventare il figlio sinistro del nodo più sinistro (e quindi minimo) del sottoalbero destro (trasformazione (i) in Fig.2.15). Viceversa, spostando il sottoalbero destro, la radice del sottoalbero spostato dovrebbe diventare il figlio destro del nodo più destro (e quindi massimo) del sottoalbero sinistro. È utile osservare, che ai fini del mantenimento della condizione di ordinamento dell'albero, è irrilevante che il nodo da cancellare sia il figlio sinistro o destro del nodo radice.

I diversi casi in cui il nodo da cancellare ha uno o due successori possono essere convenientemente unificati in un'unica implementazione il cui funzionamento è illustrato in Fig.2.16:

```
delete(struct btree ** ptrptr)
/* cancella un nodo su albero binario di ricerca in forma collegata.
   ptrptr contiene l'indirizzo del puntatore al nodo d da cancellare.
   Se *ptrptr==NULL allora il nodo da cancellare non esiste.
*/
{
    struct btree * tmp_ptr;

    if(*ptrptr!=NULL)
    {   tmp_ptr=*ptrptr;
```

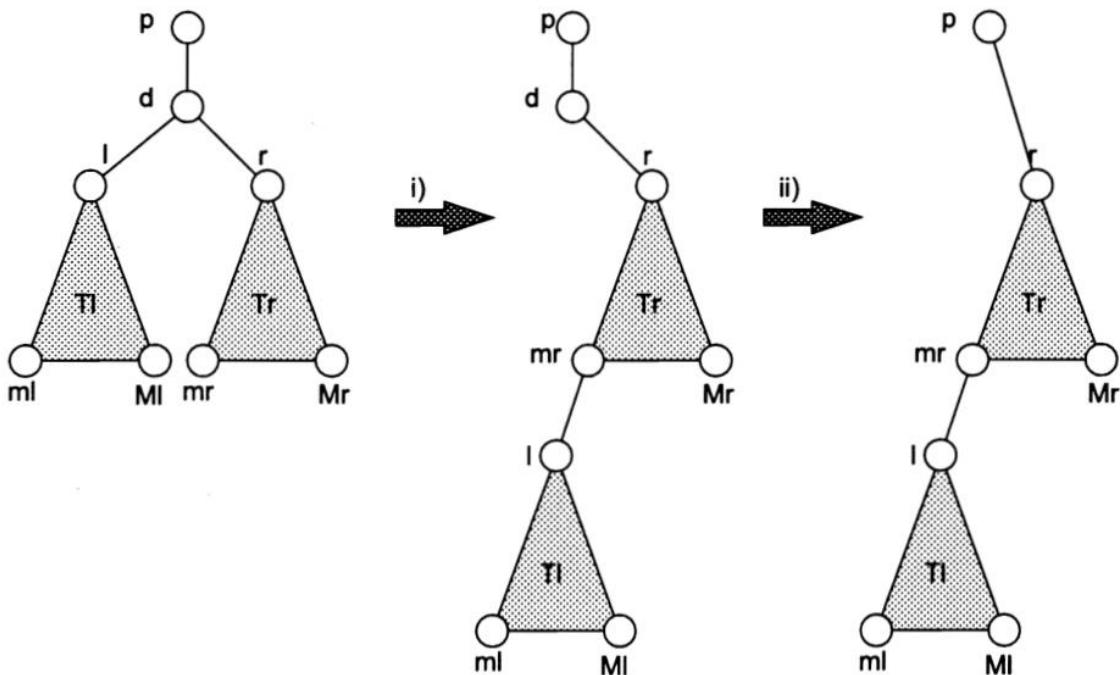


Figura 2.15: Schema della operazione di cancellazione di un nodo su un albero binario di ricerca. p denota un nodo, d un suo successore che deve essere cancellato, Tl e Tr i sottoalberi sinistro e destro di d . Nel caso generale in cui d ha due successori, il sottoalbero sinistro viene agganciato al nodo più sinistro del sottoalbero destro (trasformazione i)) e poi il puntatore da p a d viene rediretto a puntare la radice del sottoalbero destro (trasformazione ii)).

```

*ptrptr=(*ptrptr)->rx_ptr;

ptrptr=&((*ptrptr)->rx_ptr);
while(*ptrptr!=NULL)
    ptrptr=&((*ptrptr)->lx_ptr);

*ptrptr=tmp_ptr->lx_ptr;

free(tmp_ptr);
}
}

```

2.2.3 Forma sequenziale

Nella forma sequenziale i nodi sono rappresentati su un vettore e la loro relazione di discendenza è codificata in modo implicito dalla posizione occupata nel vettore: la radice è memorizzata in posizione 0; il figlio sinistro del nodo in posizione k è codificato in posizione $left(k) = 2k + 1$; il figlio destro in posizione $right(k) =$

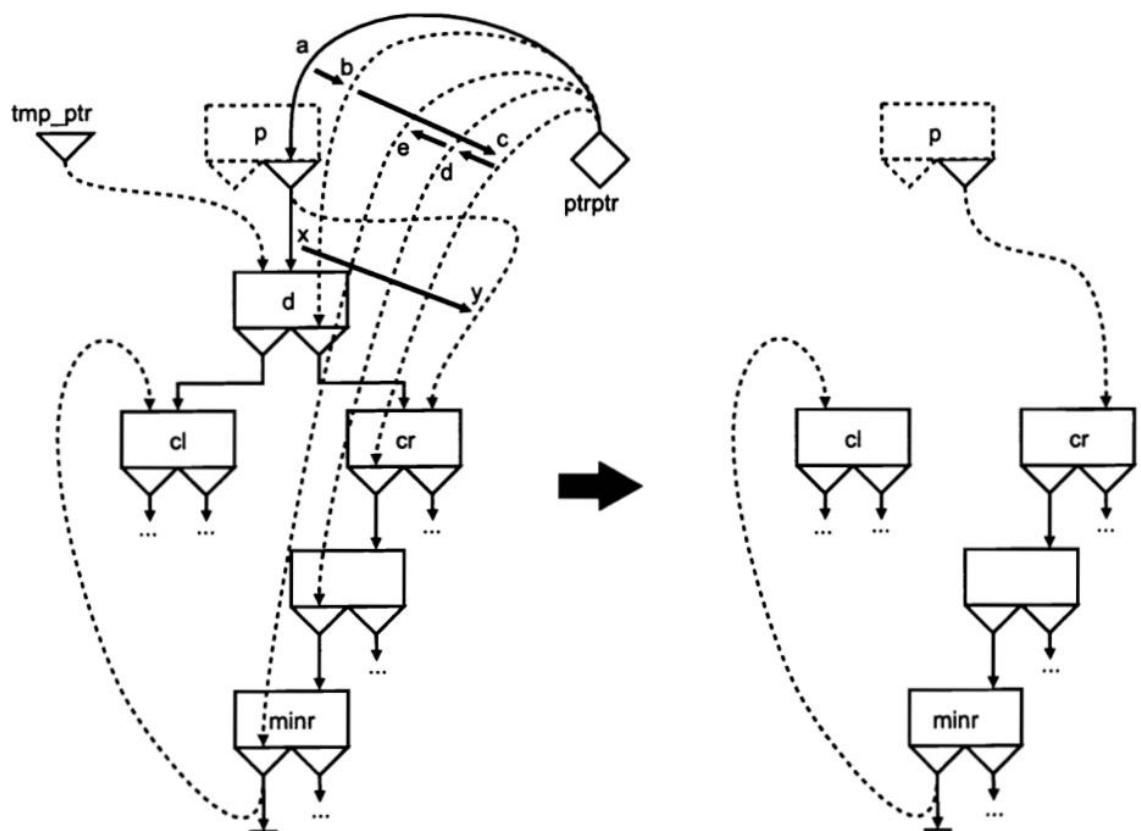


Figura 2.16: Procedura della funzione `delete` che realizza la cancellazione di un nodo su un albero binario di ricerca in forma collegata con puntatori. Inizialmente `ptrptr` contiene l'indirizzo di un puntatore (a) che contiene l'indirizzo del nodo d da cancellare (x); si osservi che il puntatore indirizzato da `ptrptr` potrebbe essere indifferentemente il campo `rx_ptr` o `lx_ptr` di un qualche nodo p oppure potrebbe essere un puntatore alla radice dell'albero. Il puntatore a d viene rediretto a puntare il figlio destro di d stesso (x→y). `ptrptr` viene rediretto a contenere l'indirizzo del puntatore alla radice del sottoalbero destro di d (a→b) e viene quindi fatto avanzare ripetutamente scendendo sempre verso sinistra nel sottoalbero destro di d (b→c→d→e) fino a contenere l'indirizzo del primo puntatore incontrato che contenga il valore NULL (e). Su tale puntatore viene copiato l'indirizzo del successore sinistro di d. `tmp_ptr` serve come variabile di appoggio per contenere l'indirizzo del nodo d.

$2k+2$. Di conseguenza il parent di un nodo in posizione $k > 0$ occupa la posizione $\text{parent}(k) = \lfloor (k-1)/2 \rfloor$.

Così facendo, gli elementi dell'albero sono disposti nel vettore in ordine breadth-first: prima la radice, poi i 2^1 nodi del livello 1, poi i 2^2 nodi del livello 2 e così via. Su un array di $\sum_{d=0}^D 2^d = 2^{D+1} - 1$ locazioni sono codificati tutti e soli i nodi di un albero binario di D livelli.

Al di là della capacità di rappresentare solo un numero prefissato di nodi, la rappresentazione ha un limite nella difficoltà di distinguere quali posizioni del vettore codificano un nodo effettivamente presente nell'albero. Il problema può essere risolto associando a ciascun nodo un flag di validità, ma questo evidentemente riduce di molto l'efficienza e l'eleganza della soluzione:

```
struct node{
    float value;
    short unsigned int is_valid
};
```

Nel caso in cui l'albero sia *completo*, ovvero quando esiste un valore M per cui i nodi validi sono tutti e soli quelli in posizione minore di M , diventa possibile evitare l'indicatore di validità e l'albero è sufficientemente codificato attraverso il numero dei nodi presenti (Fig.2.17). Questo tipo di rappresentazione trova una importante applicazione nell'algoritmo di ordinamento heapsort, discusso nel capitolo 2.5.5.

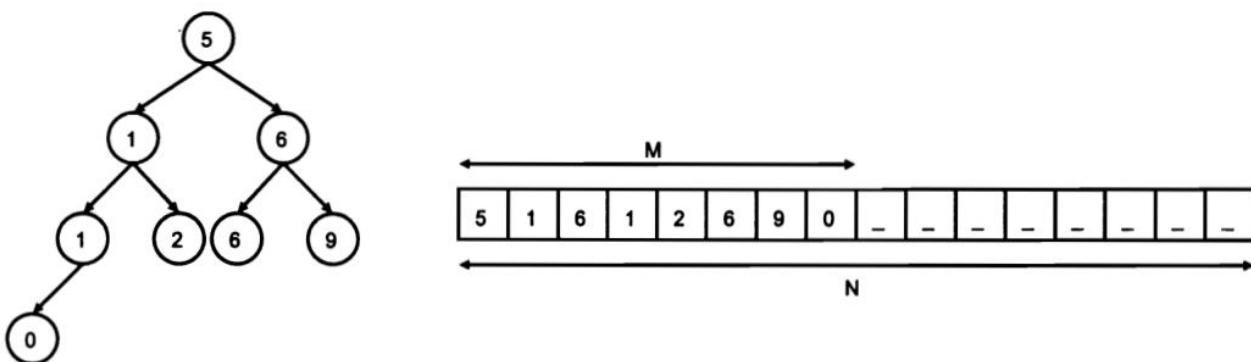


Figura 2.17: Un albero binario di ricerca completo e la sua rappresentazione in forma sequenziale. La scansione del vettore visita l'albero in ordine breadth-first. L'indice M identifica gli elementi validi nel vettore.

Sotto l'assunzione che l'albero binario sia completo, riportiamo l'implementazione delle operazioni di inizializzazione, visita, inserimento e cancellazione. Per mantenere la condizione di completezza l'inserimento (la cancellazione) deve necessariamente essere effettuata nella prima posizione libera (nell'ultima posizione

occupata) lungo il vettore. La visita è implementata in due forme. La scansione diretta del buffer produce in modo semplice ed efficiente una visita in ordine breadth-first. La visita simmetrica si realizza in forma ricorsiva ma richiede un parametro aggiuntivo che codifichi l'indice del sottoalbero da visitare; per nascondere l'inizializzazione del parametro nella prima chiamata viene implementata una funzione di facciata. Per la ragione dell'efficienza, le funzioni `left()`, `right()` e `parent()` dovrebbero più convenientemente essere implementate come macro o espande nel codice.

```

struct btree
{
    int size;
    int used;
    float * buffer;
};

void init(struct btree * ptr, int size)
/* setta la dimensione del buffer su cui sono memorizzati i valori
   e lo alloca dinamicamente;
   inizializza a 0 il numero di elementi usati
*/
{
    ptr->size=size;
    ptr->buffer=(float *)malloc(ptr->size*sizeof(float));
    ptr->used=0;
}

void visit(struct btree *ptr)
/* visita in ordine breadth first */
{
    int count;

    for(count=0;count<ptr->used;count++)
        printf("%f", ptr->buffer[count]);
}

void visit_sym(struct btree * ptr, int root)
/* visita simmetrica, in forma ricorsiva */
{
    if(root<=ptr->used)
    {
        visit(ptr, left(root));
        printf("%f", ptr->buffer[root]);
        visit(ptr, right(root));
    }
}

int left(int k)
/* restituisce la posizione del figlio sinistro di k */
{
    return 2k+1;
}

int right(int k)

```

```

/* restituisce la posizione del figlio destro di k */
{   return 2k+2;
}

int right(int k)
/* restituisce la posizione del padre di k;
   se k \e la radice restituisce un valore negativo */
{   if(k>0)
    return (k-1)/2;
   else
    return -1;
}

void visit_sym_facade(struct btree *ptr)
/* funzione di facciata sulla visita simmetrica */
{
   visit_sym(ptr, 0);
}

Boolean insert(struct btree * ptr, float value)
/* inserimento nella prima posizione libera;
   restituisce FALSE se il buffer \e pieno */
{
   if(ptr->used<ptr->size)
   {   ptr->buffer[ptr->used]=value;
       ptr->used++;
       return TRUE;
   }else
       return FALSE;
}

```

2.2.4 Forma collegata con indici

La forma collegata con indici ricalca sostanzialmente la struttura della rappresentazione collegata con indici di una lista, salvo che in questo caso il record che descrive un nodo include due indici left e right al posto del singolo indice next:

```

struct record{
    float value;
    int left;
    int right;
};

struct btree{
    int size;
    int free;
    int free;
    struct record * buffer;
};

```

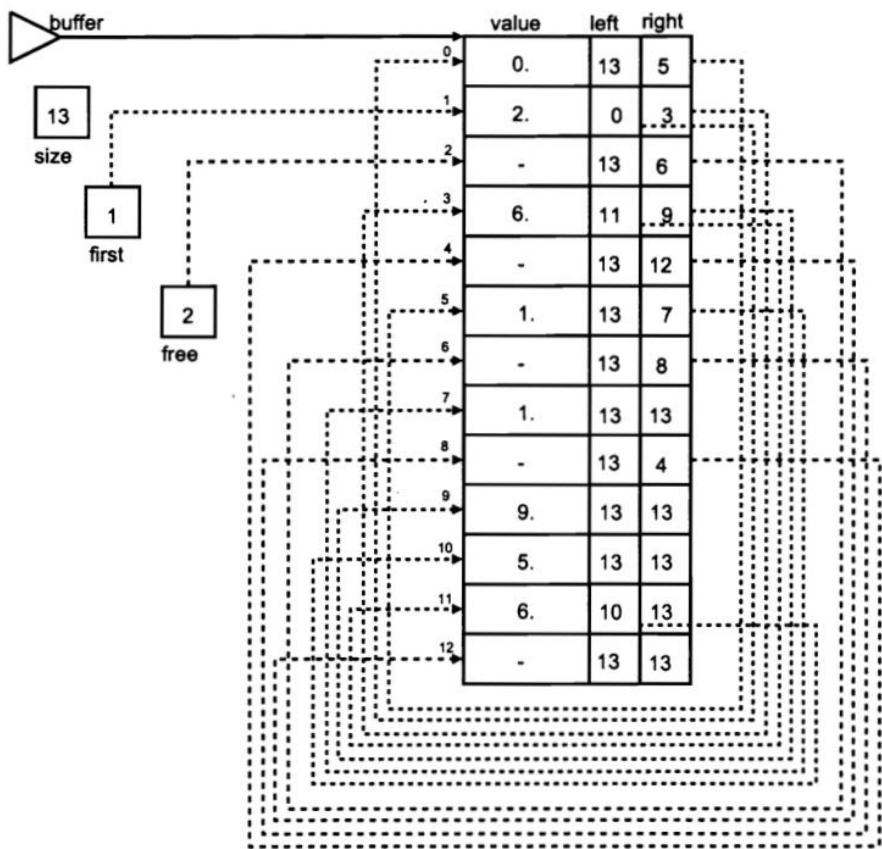


Figura 2.18: La rappresentazione in forma collegata con indici dell'albero binario di Fig.2.12a. I records liberi sono collegati in forma lineare avendo ciascuno un unico successore.

Anche in questo caso gli elementi liberi sono mantenuti collegati a partire dall'indice `free` per permettere il reperimento di un record libero in tempo costante (vedi Fig.2.18). La concatenazione tra gli elementi liberi è mantenuta in forma lineare, eseguendo tutte le operazioni di inserimento e cancellazione in testa per avere un costo di tempo costante. Come conseguenza, in tutti gli elementi liberi l'indice `right` ha sempre il valore nullo codificato con un valore illegale che tipicamente è la lunghezza del buffer di memorizzazione.

Per illustrare il concetto riportiamo la sola implementazione della operazione di inizializzazione che alloca il buffer e concatena gli elementi liberi (vedi Fig.2.19):

```
void init(struct btree * ptr, int size)
{
    ptr->size=size;
    ptr->buffer=(float *)malloc(ptr->size*(sizeof(float)));
    ptr->first=ptr->size;
    for(count=0;count<ptr->size;count++)
    {
        ptr->buffer[count].right=ptr->size;
```

```
ptr->buffer[count].left=ptr->count+1;  
}  
}
```

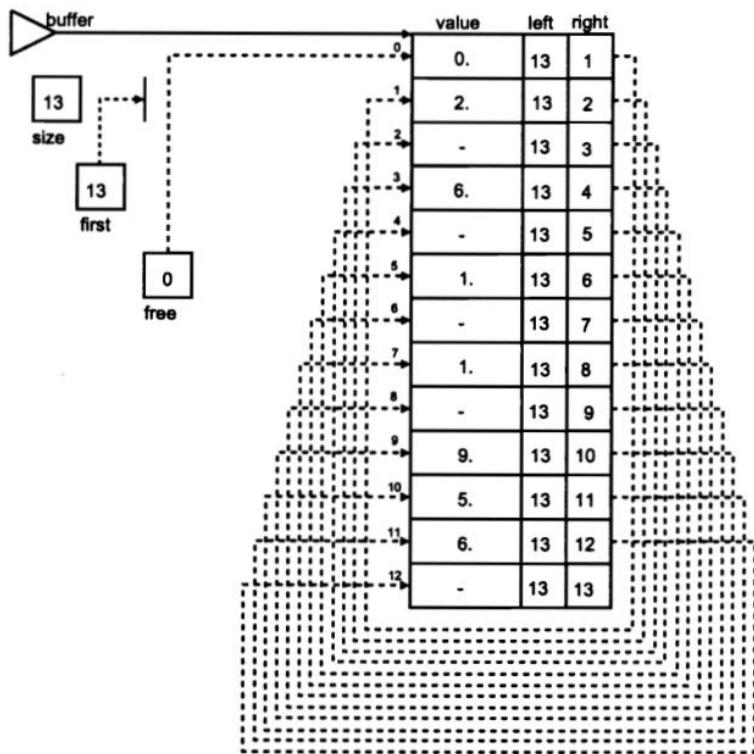


Figura 2.19: La configurazione all'inizializzazione di un albero binario in forma collegata con indici.

2.3 Costo di esecuzione e complessità

Il costo di esecuzione di un algoritmo quantifica le risorse necessarie per l'esecuzione dell'algoritmo stesso. Concretamente le risorse più rilevanti per qualificare la domanda di un algoritmo sono il numero di operazioni eseguite e la quantità di variabili allocate, che nella pratica si traducono in tempo di calcolo e spazio di memoria. La trattazione è focalizzata sul tempo di esecuzione. Non perché il problema dello spazio sia di minore rilievo, ma perché i meccanismi impiegati nel caratterizzare il costo nel tempo e nello spazio sono sostanzialmente analoghi, e quelli relativi al tempo si esemplificano in modo più naturale in riferimento ad algoritmi di ricerca e ordinamento.

2.3.1 Il modello di costo

Il tempo richiesto per l'esecuzione di un algoritmo non può essere quantificato misurando la durata dell'esecuzione né il numero di istruzioni eseguite nella CPU. In entrambi i casi la valutazione diventerebbe dipendente da un numero di fattori specifici relativi alle caratteristiche del calcolatore, del sistema operativo, del linguaggio di programmazione, del compilatore, dei dettagli dell'implementazione.

Ci serve invece riferire la valutazione ad una misura che sia sufficientemente astratta da permettere il confronto tra algoritmi progettati in tempi diversi per essere eseguiti su diversi sistemi di calcolo. E anche ci serve che la valutazione possa essere effettuata prima della implementazione dell'algoritmo, per le ragioni della leggerezza e dell'iterazione che sono inerenti al processo di progettazione. Per questa ragione la misura del costo effettivo viene sostituita con la sua valutazione rispetto ad un modello di costo.

Non esiste un unico possibile modello di costo. In generale la sua scelta attiene al contrasto tra la precisione e la generalità della valutazione. Per gli obiettivi di questo testo è conveniente assumere il modello più generale e meno preciso, che poi è anche il più semplice:

- il costo deriva interamente dagli operatori che figurano nelle espressioni. In particolare si trascurano il costo per dereferenziare un indirizzo, o per selezionare un elemento in un array o in una struttura, e si trascura il costo per eseguire una chiamata a funzione o per allocare parametri formali o variabili locali.
- il costo di qualsiasi operatore è uguale a 1.

Il modello ha dei limiti, che però rispondono ai requisiti dell'astrazione: una somma costa meno di un prodotto, ma questo dipende dall'HW; una chiamata a funzione richiede operazioni sullo stack e un salto del program counter, ma questo dipende

dal linguaggio e dalla tecnica di legame dei parametri; un riferimento complesso a variabile comporta il calcolo di un indirizzo, ma questo dipende dall'HW e dal suo assembler; una soluzione ricorsiva in generale costa più del suo equivalente iterativo, ma questo dipende dallo stile di implementazione dell'algoritmo.

Dove esistono specifici obiettivi di comparazione, può diventare conveniente ridurre la capacità di astrazione a vantaggio della precisione della stima. Ad esempio è possibile assumere che un prodotto pesi 4 volte una somma, oppure che il costo delle somme e degli assegnamenti sia trascurabile rispetto a quello di una radice quadrata e che quello di una radice sia pari al numero di cifre significative con cui essa viene calcolata.

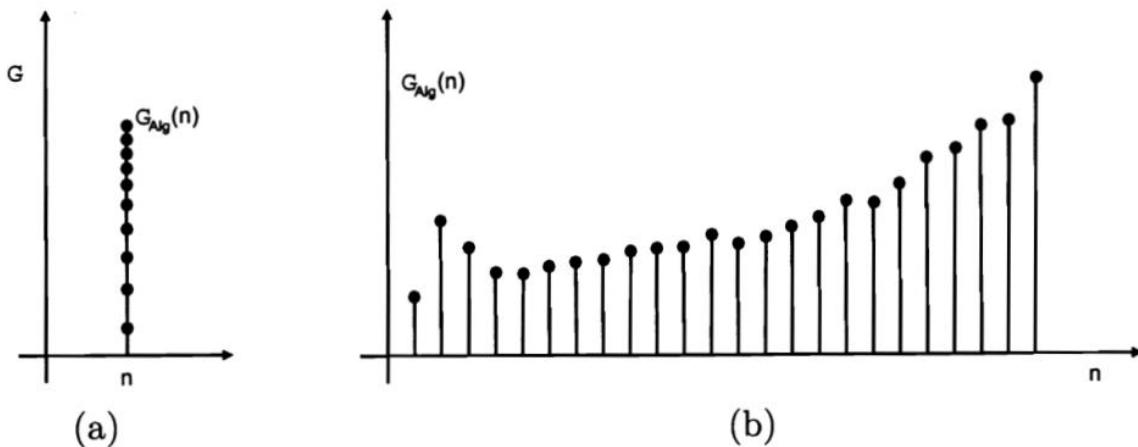


Figura 2.20: (a) Fissata una dimensione n , diversi datasets di dimensione n risultano in un diverso costo $\Gamma_{Alg}(n, D_n)$; con $\Gamma_{Alg}(n)$ denotiamo il limite superiore. (b) $\Gamma_{Alg}(n)$ varia con n .

In generale, assegnato un modello di costo Γ , un algoritmo Alg e un insieme D_n di n dati di ingresso per l'algoritmo, il modello determina in modo univoco un valore di costo $\Gamma_{Alg}(n, D_n)$. Il costo di esecuzione dipende evidentemente dalla dimensione del dataset di ingresso e dai valori specifici del dataset stesso: è probabile (anche se non necessario) che una ricerca su 128 elementi costi più di una ricerca su 64 elementi; in entrambi i casi, se il valore cercato è il primo, il costo risulterà probabilmente minore che se il valore cercato non esiste nell'insieme. Comunque sia, per ogni dimensione n del dataset esiste un limite superiore, eventualmente $+\infty$, al costo realizzato dall'algoritmo al variare dei dati in un set di dimensione n (Fig.2.20a):

$$\Gamma_{Alg}(n) = \text{Sup}_{D_n} \Gamma_{Alg}(n, D_n) \quad (2.5)$$

Possiamo allora qualificare il costo di un algoritmo attraverso una funzione che associa a ogni dimensione n del dataset un costo massimo $\Gamma_{Alg}(n)$ realizzato da Alg al variare dei datasets di dimensione n (Fig.2.20b).

2.3.2 La complessità di un algoritmo

La complessità è una stima sul costo massimo di un algoritmo $\Gamma_{Alg}(n)$: si dice che un algoritmo Alg ha complessità $\Theta(f(n))$ e si scrive $C(Alg) = \Theta(f(n))$ se esistono due fattori moltiplicativi c_1 e c_2 che scalano la funzione $f(n)$ in modo da ottenere due funzioni $c_1 \cdot f(n)$ e $c_2 \cdot f(n)$ che stimano dall'alto e dal basso il costo massimo $\Gamma_{Alg}(n)$ da un qualche \bar{n} in poi (Fig.2.21). Formalmente:

$$\exists c_1, \exists c_2, \exists \bar{n} . \forall n > \bar{n} \quad c_1 \cdot f(n) \geq \Gamma_{Alg}(n) \geq c_2 \cdot f(n) \quad (2.6)$$

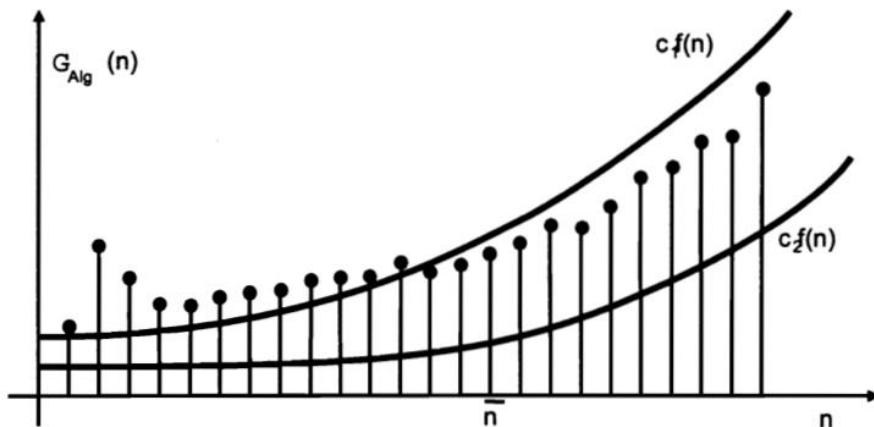


Figura 2.21: La complessità fornisce una stima sul costo massimo per l'esecuzione di un algoritmo al variare della dimensione del dataset.

Sulla complessità si applicano alcune proprietà algebriche per permettono passaggi di calcolo simbolico. La funzione $f(n)$ si dice dello stesso ordine di grandezza di $g(n)$, e si scrive $\Theta(f(n)) = \Theta(g(n))$, se esistono due fattori moltiplicativi c_1 e c_2 tali che da un certo \bar{n} in poi $c_1 \cdot f(n)$ e $c_2 \cdot f(n)$ stimano dall'alto e dal basso la funzione $g(n)$:

$$\exists c_1, c_2, \bar{n} . \forall n > \bar{n} \quad c_1 \cdot f(n) \geq g(n) \geq c_2 \cdot f(n) \quad (2.7)$$

Viceversa, si dice che $f(n)$ domina $g(n)$, e si scrive $\Theta(f(n)) > \Theta(g(n))$, se, comunque scalata la funzione $g(n)$, la funzione $f(n)$ la supera da un qualche \bar{n} in poi

$$\forall c_2 \exists \bar{n} . \forall n > \bar{n} \quad f(n) \geq c_2 \cdot g(n) \quad (2.8)$$

È facile verificare le seguenti proprietà che permettono di applicare al calcolo della complessità le semplificazioni del calcolo simbolico che sono usuali nel calcolo degli ordini di grandezza delle funzioni:

i) se $\Theta(f(n)) = \Theta(g(n))$ allora $C(Alg) = \Theta(f(n)) \leftrightarrow C(Alg) = \Theta(g(n))$;

- ii) se $\Theta(f(n)) > \Theta(g(n))$ allora $C(Alg) = \Theta(f(n)+g(n)) \leftrightarrow C(Alg) = \Theta(f(n))$;
 iii) per qualsiasi fattore di scala $c > 0$, $\Theta(f(n)) = \Theta(c \cdot f(n))$.

La possibilità di applicare alla complessità le equivalenze del calcolo simbolico degli ordini di grandezza riduce la varietà degli ordini che capita di manipolare nella analisi di complessità.

i) Un algoritmo ha complessità di tempo costante ($C(Alg) = \Theta(1)$) se il numero di operazioni per la sua esecuzione è invariante rispetto al numero di elementi nel dataset.

ii) Il minimo ordine di grandezza superiore alla costante è il logaritmo ($C(Alg) = \Theta(\ln_2(n))$), poi ci sono i polinomi ($C(Alg) = \Theta(n^\alpha)$) ordinati secondo il grado ($\forall \alpha_1 > \alpha_2 \quad \Theta(n_1^\alpha) > \Theta(n_2^\alpha)$), e poi le esponenziali, ($C(Alg) = \Theta(2^{\beta \cdot n})$) ordinate secondo la ragione dell'eponente ($\forall \beta_1 > \beta_2 \quad \Theta(2^{\beta_1 \cdot n}) > \Theta(2^{\beta_2 \cdot n})$).

iii) È noto dall'analisi delle funzioni che qualsiasi polinomio domina il logaritmo e qualsiasi esponenziale domina qualsiasi logaritmo (i.e. $\Theta(1) < \Theta(\ln_2(n)) < \Theta(n^\alpha) < \Theta(2^{\beta \cdot n}) \quad \forall \alpha, \beta > 0$).

iv) È rilevante anche il caso dell'ordine di grandezza del fattoriale, che si può ridurre all'esponenziale attraverso la formula di Stirling ($n! \simeq (\frac{n}{e})^n = 2^{n \cdot \ln_2(n) - n} \simeq 2^{n \cdot \ln_2(n)}$). Si noti che questo indica che $n!$ è peggiore di 2^n ma migliore di 2^{n^2} .

2.4 Algoritmi di ricerca

Il problema della ricerca è quello di decidere un test di inclusione: se D è uno spazio sul quale è definita una relazione di uguaglianza, che denotiamo con $==$, assegnato un multi-insieme V di N elementi in D (i.e. $V \subseteq D^N$) e un valore target $v \in D$, il problema della ricerca consiste nel decidere se esiste un elemento $V_i \in V$ tale che $v == V_i$.

In particolare ci interessa considerare qui il caso in cui i dati sono in memoria primaria, ovvero il caso in cui il costo dominante della ricerca è determinato dalle operazioni di manipolazione dei dati piuttosto che dal loro recupero da una memoria secondaria.

2.4.1 Ricerca sequenziale

Se non esistono ipotesi sui valori di V , l'unica possibilità è quella di controllarli uno ad uno, fino ad averli esauriti o fino ad aver trovato il target: per qualunque algoritmo che tentasse di venire ad una conclusione prima di avere esaminato tutti i valori sarebbe possibile identificare un dataset di ingresso che condurrebbe l'algoritmo ad una decisione errata. La complessità è evidentemente $\Theta(N)$. Per arrivarci in maniera formale si osservi che con un costo costante si compara il target contro uno degli elementi, dopodiché il problema si riduce a proseguire la ricerca sui rimanenti $N - 1$:

$$\Gamma_{bin}(N) = \begin{cases} c_1 + \Gamma_{bin}(N - 1) & \text{se } N > 0 \\ c_2 & \text{se } N == 0 \end{cases} \quad (2.9)$$

la cui soluzione è nella forma $\Gamma_{seq}(N) = c \cdot N$ per cui $C(seq) = \Theta(N)$.

L'implementazione è immediata:

```
typedef int Boolean;
#define TRUE 1
#define FALSE 0

#define PRECISION .000001
struct data{
    float key;
    ...
};

Boolean sequential_search(struct data * V, int N, struct data target)
{
    Boolean found;
    int count;
```

```

found=FALSE;
count=0;
while(found==FALSE && count<N)
{   if(is_equal(V[count],target)==TRUE)
    found=TRUE;
else
    count++;
}
return found;
}

Boolean is_equal(struct data a, struct data b)
{
if(a.key-b.key<PRECISION*a.key && -PRECISION*a.key<a.key-b.key)
    return TRUE;
else
    return FALSE;
}

Boolean is_lower(struct data a, struct data b)
{
if(a.key<b.key)
    return TRUE;
else
    return FALSE;
}

```

Si osservi che per via della precisione finita, il test di uguaglianza è sostituito con la verifica che la distanza relativa sia minore della precisione di macchina. Si osservi anche che l'algoritmo potrebbe essere ottimizzato espandendo le funzioni `is_equal` e `is_lower` nel modulo chiamante per evitare la copia dei parametri sullo stack di sistema.

2.4.2 Ricerca binaria

Se i valori in V sono ordinati (i.e. $V_{i+1} \geq V_i$ per $i = 0, N - 2$), allora l'esito di un solo test può escludere più di un elemento di V dallo spazio della ricerca.

Prima di procedere è utile soffermarsi sul significato dell'ipotesi. Essa sottende che nello spazio dei valori D sia definita una relazione di ordine totale, ragionevolmente denotata come \geq , e che gli elementi di V possano essere numerati in modo univoco come V_i con $i = 0, N - 1$. Mentre nel caso di tipi elementari la relazione di ordine usualmente corrisponde alla semantica dell'operatore \geq , in dati complessi (strutture) essa potrebbe essere definita in maniere diverse, su una o più chiavi o addirittura su una loro combinazione.

Se oltre all'ipotesi di ordinamento possiamo anche assumere che i valori sono

memorizzati su un vettore, diventa conveniente applicare una ricerca binaria (o dicotomica): il target viene comparato contro l'elemento in posizione mediana (i.e. $V[N/2]$); se è uguale la ricerca termina con successo; se è maggiore la ricerca prosegue nella metà destra del vettore; se è minore prosegue nella metà sinistra. Quando la ricerca viene applicata su un vettore di lunghezza 0 è possibile concludere che il target non è presente nell'insieme.

Il caso pessimo si realizza quando il test di uguaglianza fallisce sempre. In tal caso il costo di esecuzione è pari a un valore costante c (speso per verificare la dimensione del vettore su cui stiamo operando la ricerca, per confrontare l'elemento mediano contro il target e per decidere su quale metà proseguire) più il costo per proseguire la ricerca sulla metà selezionata:

$$\Gamma_{bin}(N) = \begin{cases} c_1 + \Gamma_{bin}(N/2) & \text{se } N > 0 \\ c_2 & \text{se } N == 0 \end{cases} \quad (2.10)$$

Da cui segue $\Gamma_{bin}(N) = c \cdot \ln_2(N)$ e quindi $C_{bin} = \Theta(\ln_2(N))$.

L'algoritmo ha una naturale implementazione in forma ricorsiva. Per evitare di ripetere dettagli già discussi, assumiamo che i dati siano di tipo intero:

```
Boolean b_search(int * V, int N, int target)
{
    if(N>0)
    {
        if(V[N/2]==target)
            return TRUE;
        else
            if(V[N/2]<target)
                return b_search(V,N/2,target);
            else
                return b_search(&V[N/2+1],N-N/2-1,target);
    }
    else
        return FALSE;
}
```

Si osservi che $N/2$ è il numero di elementi che precedono l'elemento $V[N/2]$, che $N-N/2-1$ è il numero di elementi che lo seguono, e che $\&V[N/2+1]$ è l'indirizzo del primo elemento successivo a $V[N/2]$. Anche per N dispari.

Una implementazione in forma iterativa si può ottenere utilizzando due indici `first` e `length` che nel corso della ricerca codificano l'indice del primo elemento e la lunghezza del sotto-vettore nel quale è possibile trovare il valore cercato:

```
Boolean b_search_iter(int * V, int N, int target)
{
    int first;
    int length;
```

```

Boolean found;

first=0;
length=N;
found=FALSE;
while(length>0)
{
    if(V[first+length/2]==target)
        found=TRUE;
    else
        if(V[first+length/2]>target)
            length=length/2
        else
            {   first=first+length/2+1;
                length=length-length/2-1;
            }
}
return found;
}

```

È utile osservare che l'ipotesi che i dati siano memorizzati su un vettore permette di attribuire un costo costante alla selezione dell'elemento in posizione mediana. In generale questo si applica a qualsiasi struttura dati che permetta l'accesso diretto a valori in posizioni intermedie, incluso il caso di una lista in forma sequenziale. Se i dati di V fossero invece memorizzati su una lista in forma collegata, l'accesso all'elemento mediano avrebbe un costo lineare e l'equazione di costo degenererebbe nella forma

$$\Gamma_{bin}(N) = \begin{cases} c_1 \cdot N/2 + \Gamma_{bin}(N/2) & se N > 0 \\ c_2 se N == 0 \end{cases} \quad (2.11)$$

Questa ha soluzione nella forma $\Gamma_{bin}(N) = c \cdot N$, che può essere conseguito con più semplicità attraverso una ricerca sequenziale.

2.4.3 Ricerca a salti

Nell'ipotesi che i dati siano in ordine, è possibile applicare una ricerca nella quale l'insieme V viene campionato saltando ripetutamente in avanti: il campionamento prosegue fino al momento in cui un campione è maggiore o uguale al valore cercato, oppure fino al momento in cui l'ultimo possibile campione è ancora minore del valore cercato. Nel secondo caso la ricerca è fallita, mentre nel primo essa viene completata eseguendo una scansione sequenziale degli elementi compresi tra gli ultimi due campioni.

L'implementazione si basa su due iterazioni successive che pilotano due indici: up viene incrementato con il passo del salto fino a che viene esaurito il vettore

oppure viene raggiunto un valore non minore del valore cercato; in questo secondo caso, un secondo indice down viene decrementato di una locazione per volta fino a raggiungere il valore cercato o un suo minorante (vedi Fig.2.22). Alcuni controlli ulteriori complicano l'algoritmo per gestire gli effetti di bordo nel caso in cui il primo elemento del vettore è già un maggiorante e nel caso in cui la dimensione del vettore non è un multiplo intero del salto:

```

Boolean j_search(int * V, int N, int target, int step)
/* ricerca a salti. step e' l'ampiezza del salto. */
{   int up;
    int down;
    Boolean equal_found;
    Boolean greater_or_equal_found;

    up=0;
    greater_or_equal_found=FALSE;
    while(up<N && greater_or_equal_found==FALSE)
    {   if(V[up]>=target)
        greater_or_equal_found=TRUE;
        else
        {   up+=step;
            if(up>N-1 && up<N-1+step)
                up=N-1;
        }
    }

    equal_found=FALSE;
    if(greater_or_equal_found==TRUE)
    {   down=up;
        while(V[down]>=target && equal_found==FALSE && down>=0)
        {   if(V[down]==target)
            equal_found=TRUE;
            else
                down--;
        }
    }
    return equal_found;
}

```

Il caso pessimo si realizza quando il campionamento arriva fino all'ultimo elemento del vettore e poi visita tutti gli elementi compresi tra l'ultimo e il penultimo campione. Se S denota l'ampiezza del salto e N l'ampiezza del dataset, questo risulta nella visita di $\frac{N}{S}$ campioni e poi nell scansione sequenziale di S elementi. Il costo ha quindi la forma:

$$\Gamma_{js}(N) = c1 \cdot \frac{N}{S} + c2 \cdot S$$

Per identificare il valore ottimo del salto S possiamo derivare rispetto ad S :

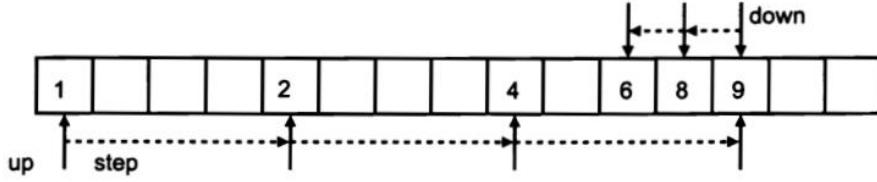


Figura 2.22: Ricerca del valore 7 secondo l'algoritmo di ricerca a salti: il vettore è campionato facendo scorrere l'indice **up** sui multipli di **step** fino a identificare un valore non minore di 7 e poi percorrendo all'indietro il vettore fino a raggiungere il valore cercato o un suo minorante. In quest'ultimo caso la ricerca conclude che il valore cercato non è presente nell'insieme.

$$\frac{\partial \Gamma_{js}(N, S)}{\partial S} = c_2 - c_1 \cdot \frac{N}{S^2}$$

La derivata ha uno zero in $S = c \cdot \sqrt{N}$ che è un minimo, come si verifica con una derivata seconda (o, più fisicamente, considerando che al peggio non esiste limite). Dunque il salto ottimo ha ampiezza $S = \sqrt{N}$ che riduce la complessità dell'algoritmo alla forma

$$C_{js} = \Theta(\sqrt{N})$$

Rispetto all'algoritmo di ricerca binaria, la ricerca a salti ha il pregio di richiedere l'accesso diretto ai soli elementi dell'insieme V che si trovano sui multipli del salto. Questo ne rende praticabile l'applicazione anche quando V è rappresentato su una lista in forma collegata. In tal caso, una lista ausiliaria di dimensione \sqrt{N} potrebbe contenere gli indirizzi dei multipli del salto, abilitando una ricerca in tempo $\Theta(\sqrt{N})$ piuttosto che $\Theta(N)$ come ottenuto con la ricerca sequenziale.

2.4.4 Esercizi

Es. 2.4.1 Si definisca una implementazione della ricerca binaria su lista sequenziale.

Soluzione: Forniamo l'implementazione in forma iterativa che risulta più naturale rispetto alle caratteristiche della lista in forma sequenziale. Per risolvere il problema, si osservi che se `ptr` è il puntatore alla lista rappresentata come nel capitolo 2.1.1, di modo che `ptr->last` e `ptr->first` sono l'indice della testa e del successore della coda, e `ptr->size` è la dimensione della lista, il numero di elementi nella lista puo' essere calcolato come `length=`

$(\text{ptr} \rightarrow \text{tail} - \text{ptr} \rightarrow \text{head} + \text{ptr} \rightarrow \text{size}) \% \text{ptr} \rightarrow \text{size}$. Inoltre, se `first` indirizza un elemento della lista, l'elemento che lo segue di $\text{length}/2$ posizioni è indirizzato da $(\text{first} + \text{length}/2) \% \text{ptr} \rightarrow \text{size}$. Adattando l'implementazione iterativa, si ottiene allora:

```
Boolean b_search_iter_seqlist(struct list * ptr, int target)
{
    int first;
    int length;
    Boolean found;

    first=ptr->last;
    length=(ptr->first-ptr->last+ptr->size)\%ptr->size;
    found=FALSE;
    while(length>0)
    {
        if(V[(first+length/2)%ptr->size]==target)
            found=TRUE;
        else
            if(V[(first+length/2)%ptr->size]>target)
                length=length/2
            else
                {   first=(first+length/2+1)%ptr->size;
                    length=length-length/2-1;
                }
    }
    return found;
}
```

□

2.5 Algoritmi di ordinamento

Assegnato un multi insieme numerato $V = \{V_n\}_{n=0}^{N-1}$ di valori in D , il problema dell'ordinamento consiste nel trovare una permutazione $n(j)$ degli indici n (ovvero una funzione biunivoca $n : [0, N - 1] \rightarrow [0, N - 1]$), tale che $j_2 \geq j_1 \rightarrow V_{n(j_2)} \geq V_{n(j_1)}$.

Come già assunto per il problema della ricerca, ci interessa considerare il caso in cui i dati sono in memoria primaria, ovvero il caso in cui il costo per il riferimento alle variabili è trascurabile rispetto a quello per il confronto e la manipolazione dei loro valori.

2.5.1 Sequential-sort

L'algoritmo Sequential Sort riduce il problema dell'ordinamento alla selezione del massimo e allo swap: sull'insieme V viene selezionato il massimo, e l'elemento che lo realizza viene swappato con l'elemento in ultima posizione; dopodiché il problema prosegue allo stesso modo sui primi $N-1$ elementi (vedi Fig.2.23). Dopo k passaggi, nelle ultime k posizioni ci sono i k elementi maggiori. Di conseguenza, dopo $N-1$ passaggi, il vettore è ordinato.

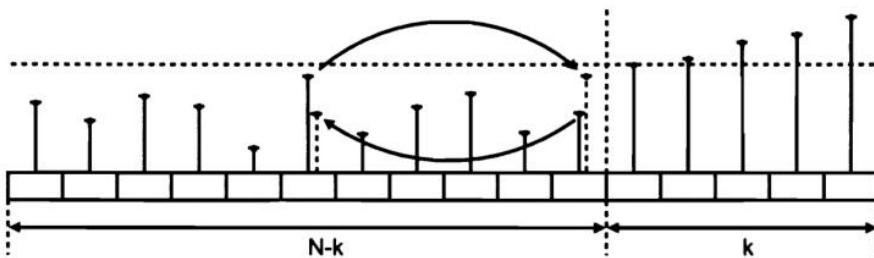


Figura 2.23: L'algoritmo sequential-sort. Dopo k iterazioni, gli ultimi k elementi sono nella loro corretta posizione. Viene selezionato il massimo sui primi $N-k$ e viene swappato con l'elemento in posizione $N-k-1$. Questo riduce il problema dalla dimensione $N-k$ alla dimensione $N-k-1$. Dopo $N-1$ iterazioni l'algoritmo termina.

L'equazione di costo ha la forma

$$\Gamma_{SeqS}(N) = \begin{cases} c_1 \cdot N + c_2 + \Gamma_{SeqS}(N-1) & \text{se } N > 1 \\ c_3 & \text{se } N == 0 \end{cases} \quad (2.12)$$

dove $c_1 \cdot N$ è il costo di selezione del massimo e c_2 è il costo dello swap. La soluzione è della forma

$$\Gamma_{SeqS}(N) = c \cdot \frac{N \cdot (N - 1)}{2}$$

e quindi

$$C_{SeqS}(N) = \Theta(N^2)$$

Riportiamo l'implementazione nel caso in cui l'insieme V sia rappresentato su un vettore. La soluzione è banale, ma permette di focalizzare l'attenzione sul meccanismo di virtualizzazione dell'ordinamento: piuttosto che modificare la posizione degli elementi nel vettore, si costruisce un vettore di permutazione che fornisce la sequenza con cui il vettore può essere visitato in ordine. Questo ha il vantaggio di non perdere l'eventuale informazione codificata nel disordine iniziale del vettore e permette una sostanziale ottimizzazione nel caso in cui i dati siano complessi e/o possano essere ordinati ripetutamente rispetto ad una molteplicità di chiavi.

```

void sequential_sort(struct data * V, int N, int * Perm)
/* costruisce in Perm il vettore di permutazione tale per cui
   k2>=k1 implica V[Perm[k2]]>=V[Perm[k1]]
*/
{
    int count;
    int count_of_max;
    int iter;

    for(count=0;count<N;count++)
        Perm[count]=count;

    for(iter=0;iter<N-1;iter++)
    {
        for(count=1,count_of_max=0;count<N-iter;count++)
            if(V[Perm[count]]>V[Perm[count_of_max]])
                count_of_max=count;
        swap(Perm,count_of_max,N-iter-1);
    }
}

```

Si osservi che l'espressione $V[Perm[count]] > V[Perm[max_index]]$ non è legale se data è un tipo strutturato. In tal caso, essa dovrebbe essere sostituita con l'espressione $V[Perm[count]].key > V[Perm[max_index]].key$, oppure con l'espressione `is_greater(V[Perm[count]], V[Perm[max_index]])` dove `is_greater` è una funzione che incapsula il confronto.

È interessante osservare un modo di ridurre il numero di confronti dell'algoritmo sostituendo la ricerca e lo swap del massimo con la ricerca e lo swap contemporaneo di massimo e minimo. Così facendo il numero di iterazioni si

dimezza mentre il costo di ciascuna iterazione aumenta. Il guadagno deriva dal fatto che la ricerca contemporanea di massimo e minimo può essere effettuata con un numero di confronti minore del doppio dei confronti richiesti per la ricerca del solo massimo: dovendo comparare x e y contro un massimo \max e un minimo \min si compara inizialmente x contro y ; solo il maggiore dei due può diventare il massimo, e solo il minore dei due può diventare il minimo. Per questo il numero di confronti richiesti per trattare x e y rispetto a \min e \max si riduce da 4 a 3:

```
if(x>y)
{   if(x>max)
    max=x;
    if(y<min)
    min=y;
} else
{   if(y>max)
    max=y;
    if(x<min)
    min=x;
}
```

L'ottimizzazione non è così rilevante nel contesto della nostra trattazione: in ultimo riduce il costo di un fattore $\frac{3}{4}$ e ovviamente non incide sull'ordine della complessità. È però indicativa del tipo di ottimizzazione fine che è possibile su un algoritmo, che noi sistematicamente omettiamo qui per il bene della semplicità.

2.5.2 Bubble-sort

L'algoritmo BubbleSort riduce il problema dell'ordinamento al confronto e lo swap di elementi successivi. L'algoritmo opera in iterazioni successive: nella k -esima iterazione un indice $count$ scandisce le posizioni da 0 a $N - k - 1$, confrontando ad ogni passo il valore di $V[count]$ contro quello di $V[count + 1]$ e permutando le posizioni dei due elementi se essi non sono in ordine. L'algoritmo termina quando nel corso di una iterazione non sono stati identificati due elementi fuori ordine. Come vedremo quantitativamente nel seguito, l'algoritmo non ha particolare pre-gio nell'efficienza, ma offre una occasione interessante per esemplificare concretamente il problema della verifica di correttezza. In generale, il problema della correttezza è convenientemente scomposto in due parti: la *correttezza parziale* è quella per cui se l'algoritmo termina, allora il risultato prodotto è corretto; la *correttezza totale* si raggiunge quando alla correttezza parziale viene aggiunta la garanzia di *terminazione*.

Nel caso del bubble sort, per come lo abbiamo formulato, la correttezza parziale è garantita in modo immediato: l'algoritmo termina quando nel corso di una iterazione non sono identificati alcuni due elementi contigui fuori ordine (i.e.

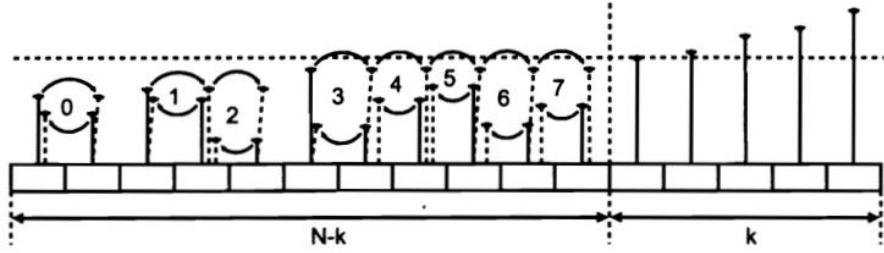


Figura 2.24: L'algoritmo bubble-sort. Dopo k iterazioni, gli ultimi k elementi sono nella loro corretta posizione. I primi $N-k$ vengono scanditi, confrontando ciascun elemento con il suo successore e swappandoli se non in ordine. Lo schema evidenzia la sequenza degli swap e la successione di valori (solido il valore iniziale e tratteggiati quelli successivi da sinistra a destra) nel corso di una iterazione. L'algoritmo termina quando in una iterazione non sono effettuati swap e comunque non oltre $N-2$ iterazioni.

$\forall n \in [0, N - 2] V[n + 1] \geq V[n]$), che garantisce che il vettore è in ordine. Il problema della verifica è scaricato completamente sulla garanzia di terminazione. Questa può essere dimostrata sulla base di due proprietà che caratterizzano in una prospettiva globale l'effetto dei confronti e delle permutazioni locali effettuate nel corso di ciascuna iterazione.

- i. Un elemento che all'inizio dell'iterazione non è preceduto da alcun maggiorante avanza fino ad incontrare il primo maggiorante; nel caso particolare in cui non ha alcun maggiorante avanza fino all'ultima posizione nel vettore.

Formalmente: se inizialmente l'elemento $V[\bar{n}]$ non è preceduto da alcun maggiorante (i.e. $\forall n \in [0, \bar{n}] V[\bar{n}] \geq V[n]$) ed è seguito da m minoranti e poi da un maggiorante oppure dal termine del vettore (i.e. $\forall n \in [\bar{n} + 1, \bar{n} + m] V[\bar{n}] > V[n] \wedge (V[\bar{n} + m + 1] \geq V[\bar{n}] \vee \bar{n} + m = N - 1)$), allora, al termine della iterazione, $V[\bar{n}]$ si trova in posizione $\bar{n} + m$.

Seguendo la sequenza delle operazioni nell'iterazione, il primo confronto che coinvolge $V[\bar{n}]$ è contro il suo predecessore, che è senz'altro uno dei valori che inizialmente avevano indice minore di \bar{n} e che quindi è minore secco di $V[\bar{n}]$; dunque non viene eseguito swap e $V[\bar{n}]$ conserva la sua posizione. Successivamente, $V[\bar{n}]$ viene comparato con il suo successore che per ipotesi è un minorante; viene quindi eseguito uno swap e $V[\bar{n}]$ prende la posizione $\bar{n} + 1$. La stessa operazione si ripete fino a quando $V[\bar{n}]$ ha raggiunto il suo primo maggiorante oppure la fine del vettore.

- ii. Un elemento che all'inizio dell'iterazione è preceduto da almeno un maggiorante, al termine dell'iterazione è arretrato di una posizione.

Formalmente: se inizialmente l'elemento $V[\bar{n}]$ è preceduto da almeno un maggiorante (i.e. $\exists n \in [0, \bar{n}] \quad V[n] > V[\bar{n}]$), allora al termine della iterazione $V[\bar{n}]$ si trova in posizione $\bar{n} - 1$.

Come conseguenza di *i*, quando $V[\bar{n}]$ viene comparato con il suo predecessore, in posizione $\bar{n} - 1$ si trova il massimo tra i valori inizialmente contenuti nelle posizioni $[0, \bar{n}]$. Per ipotesi, tale massimo è maggiore di $V[\bar{n}]$ che viene quindi swappato indietro.

Come corollario di *i*, al termine della k -esima iterazione, le ultime k posizioni del vettore contengono i k maggiori elementi nel loro corretto ordine. Questo implica che entro $N - 2$ iterazioni, gli ultimi $N - 1$ elementi sono nella loro corretta posizione e quindi l'intero vettore è ordinato. Si osservi che, sempre per conseguenza della proprietà *i*, la k -esima iterazione può essere limitata alle sole prima $N - k$ posizioni.

Grazie a questo il costo di esecuzione su N elementi può essere espresso come il costo dell'iterazione su N elementi, più il costo per eseguire l'ordinamento sui primi $N - 1$. A sua volta il costo dell'iterazione su N elementi comporta $N - 1$ confronti e, nel caso pessimo $N - 1$ operazioni di swap:

$$\Gamma_{BubS}(N) = \begin{cases} (s + c) \cdot N + \Gamma_{BubS}(N - 1) & \text{se } N > 1 \\ c_3 & \text{se } N == 0 \end{cases} \quad (2.13)$$

dove s è il costo di uno swap e c il costo di un confronto. Integrando l'equazione, il risultato ha la forma:

$$\Gamma_{BubS}(N) = (s + c) \frac{(N - 1)(N - 2)}{2}$$

per cui la complessità ha ancora la forma già ottenuta per il sequential sort:

$$C_{BubS}(N) = \Theta(N^2)$$

Pur realizzando la stessa complessità, tra gli algoritmi sequential sort e bubble sort esistono sostanziali differenze.

Mentre il sequential sort ha un costo invariante rispetto ai dati, il bubble sort può terminare senza avere compiuto tutte le $N - 2$ iterazioni. Questo risulta in una riduzione del numero di iterazioni. Come svantaggio, mentre il sequential sort richiede sempre $N - 1$ swaps, il bubble sort può richiederne $(N - 1)(N - 2)$. Questo rende vitale per il bubble sort l'adozione di una virtualizzazione dell'ordinamento per ridurre l'impatto del costo di swap.

```
void BubbleSort(struct data * V, int N, int * Perm)
```

```

{ int iter;
Boolean iter_without_swap;
Boolean swap_found;

iter=0;
iter_without_swap=FALSE;
while(iter_without_swap==FALSE)
{ for(count=0, swap_found=FALSE; count<N-iter-1; count++)
  { if(V[Perm[count]]<V[Perm[count+1]])
    { swap(Perm, count, count+1);
      swap_found=TRUE;
    }
  }
  if(swap_found==FALSE)
    iter_without_swap=TRUE;
  else
    iter++;
}
}

```

Si osservi che nella descrizione proposta l'algoritmo è fortemente asimmetrico: in accordo con le proprietà *i.* e *ii.*, i valori pesanti scendono velocemente, mentre quelli leggeri salgono al massimo di una posizione a iterazione.

L'asimmetria si manifesta nel diverso modo con cui l'algoritmo tratta il riordinamento successivo alla perturbazione di un vettore ordinato. Si consideri il caso in cui su un vettore ordinato V viene modificato il solo valore $V[k]$. Se $V[k]$ è troppo pesante per la posizione che occupa (i.e. $V[k] > V[k+1]$), il bubblesort completa l'ordinamento in due iterazioni: nella prima porta il valore nella sua corretta posizione e nella seconda verifica che il vettore è in ordine. Nel caso opposto in cui $V[k]$ è troppo leggero (i.e. $V[k-1] > V[k]$), allora l'algoritmo spende una iterazione per ogni posizione che $V[k]$ deve risalire; nel caso limite in cui $V[k]$ è il minimo sul vettore e k è l'ultima posizione, l'algoritmo realizza il costo massimo di N iterazioni. La asimetria può essere invertita invertendo l'ordine di scansione del vettore in ciascuna singola iterazione.

Il problema può essere superato alternando la direzione delle iterazioni: nella prima si confrontano gli elementi da 0 a $N-2$, nella seconda si confrontano gli elementi da $N-3$ a 1, nella terza da 2 a $N-4$, e così via. Con tale schema, l'algoritmo garantisce di riordinare un vettore perturbato in tempo $\Theta(N \cdot k)$ dove k è il numero di elementi perturbati.

2.5.3 Merge-sort

L'algoritmo di mergesort riduce il problema dell'ordinamento al problema della fusione: due metà del vettore sono ordinate separatamente e poi sono fuse in modo da ottenere un vettore ordinato globalmente (Fig.2.25).

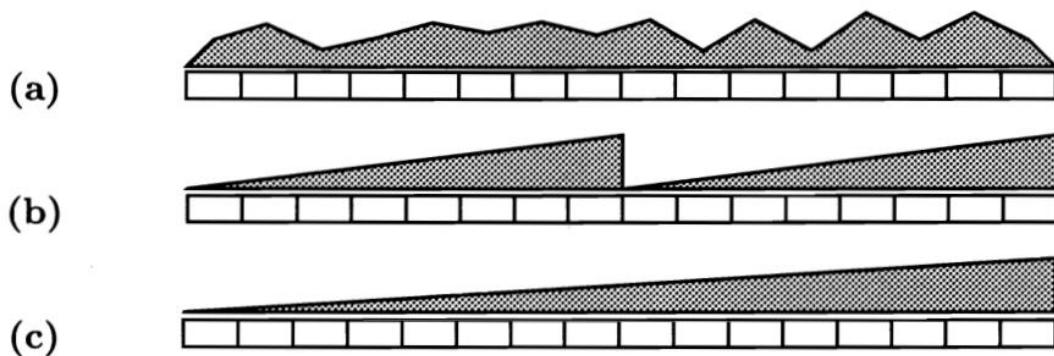


Figura 2.25: L'algoritmo merge sort. Il vettore viene partizionato in due semi-vettori che sono ordinati separatamente e poi fusi per ottenere un ordinamento globale.

Fusione

Il problema della fusione consiste nell'ordinare un vettore partizionato in due semi-vettori ciascuno dei quali è inizialmente ordinato al suo interno.

Il semi-vettore sinistro viene inizialmente copiato su un semi-vettore di appoggio. Viene poi ripetitivamente selezionato e trasferito sul vettore complessivo il minimo del semi-vettore di appoggio e di quello destro. L'algoritmo termina quando tutti gli elementi del vettore di appoggio sono stati copiati sul vettore complessivo (vedi Fig.2.26).

L'algoritmo esegue in tempo lineare rispetto alla lunghezza N del vettore complessivo: ad ogni selezione e confronto tra due minimi, il vettore complessivo cresce di un elemento, per cui l'algoritmo termina dopo un numero massimo di N selezioni; d'altra parte, essendo i due semi-vettori ordinati, la selezione e il confronto dei loro minimi avviene in tempo costante. Al costo di selezione e copia dai due semi-vettori si aggiunge il costo iniziale per copiare il semi-vettore sinistro nell'area di appoggio, che comunque avviene in tempo lineare.

L'implementazione si basa su due indici l ed r che realizzano un invarianto: nel corso di tutta la computazione, l ed r indicano il numero di elementi inizialmente contenuti nel semi-vettore di appoggio e in quello destro che sono già stati selezionati e riportati nella loro corretta posizione nel vettore complessivo. Come corollario, $l+r$ è l'indice del prossimo elemento da costruire nel vettore complessivo (vedi Fig.2.26). Inizialmente assumiamo $l=r=0$. Ad ogni iterazione gli elementi in posizione l ed r nel semivettore di appoggio e in quello destro sono comparati tra loro, il minore tra i due viene copiato in posizione $l+r$ nel vettore finale, e l'indice l o r viene incrementato a seconda di quale dei due sia stato selezionato. La terminazione dell'algoritmo è in qualche modo asimmetrica, risentendo in questo del diverso trattamento applicato al semivettore sinistro rispetto a quello

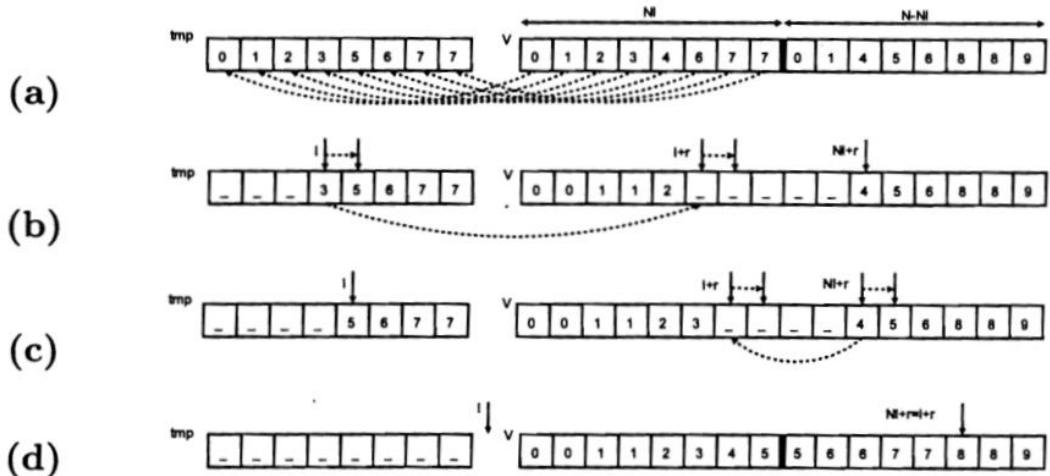


Figura 2.26: L'algoritmo di fusione riceve in ingresso un vettore le cui parti sinistra e destra sono separatamente ordinate. (a) Il semivettore sinistro viene inizialmente copiato su un vettore di appoggio. (b-c-d) Nel corso della computazione, due indici l ed r identificano il minimo sul semivettore sinistro e su quello destro. (b-c) Ripetutamente il minore tra i minimi del semivettore sinistro e destro viene selezionato e copiato nella sua posizione ordinata. (d) Quando il semivettore sinistro è esaurito il vettore è in ordine.

destro. Se il semivettore destro si esaurisce prima del semivettore di appoggio (il che capita se e solo il semivettore sinistro contiene inizialmente il massimo assoluto nel vettore) allora l'algoritmo si completa trasferendo in posizione tutti e soli gli elementi che ancora non sono stati selezionati nel semi-vettore di appoggio. Se invece si esaurisce prima il semivettore sinistro, allora l'algoritmo può terminare senza ulteriore computazione perché gli elementi residui del semi-vettore destro sono già in ordine e nella loro corretta posizione nel vettore complessivo.

```

void merge(struct data * V, int N, int Nl)
/* esegue la fusione sui due semi-vettori di V di lunghezza Nl e N-Nl.
   Per la computazione alloca e poi libera l'area di appoggio necessaria */
{
    int l;
    int r;
    struct data * tmp;
    int count;

    tmp=(struct data *)malloc(Nl*sizeof(struct data));
    for(count=0;count<Nl;count++)
        tmp[count]=V[count];

    l=0;
    r=0;

```

```

while(l<Nl && r<N-Nl)
{
    if(tmp[l]<V[Nl+r])
        { V[l+r]=tmp[l];
          l++;
        }
    else
        { V[l+r]=V[Nl+r]
          r++;
        }
}
while(l<Nl)
{
    V[l+r]=tmp[l];
    l++;
}

free(tmp);
}

```

Riduzione dell'ordinamento alla fusione

Una volta acquisita una soluzione di tempo lineare per il problema della fusione, torniamo al problema di ordinamento per fusione. Ne facciamo una occasione concreta per mostrare come partizionando i dati da trattare sia possibile ridurre la complessità di un problema nel quale il costo di soluzione cresce con un ordine di grandezza che eccede quello lineare. Il principio è spesso menzionato come divide et impera.

Un algoritmo banale come il sequential-sort risolve il problema dell'ordinamento in tempo N^2 . Questo significa che se la dimensione del vettore da ordinare raddoppia, il tempo per il suo ordinamento quadruplica ($(2N)^2 = 4N^2$). Cercando di vedere il bicchiere mezzo pieno, questo suggerisce l'osservazione inversa: se ordiniamo separatamente la prima e la seconda metà del vettore il tempo complessivo dimezza ($(\frac{N}{2})^2 + (\frac{N}{2})^2 = \frac{N^2}{2}$). Ovviamente, una volta che abbiamo ordinato le due metà non abbiamo finito il lavoro, ma con l'algoritmo di fusione possiamo completarlo in tempo N . In sostanza, se ordiniamo le due metà del vettore con un algoritmo banale e poi le fondiamo, paghiamo un costo

$$\Gamma(N) = \Gamma_{seq}(\frac{N}{2}) + \Gamma_{seq}(\frac{N}{2}) + \Gamma_{merge}(N) = (\frac{N}{2})^2 + (\frac{N}{2})^2 + N = \frac{N^2}{2} + N \quad (2.14)$$

Per N grande questo può essere un bel guadagno rispetto al costo N^2 richiesto per eseguire il sequential-sort sul vettore complessivo. Tuttavia l'ordine di grandezza della complessità non cambia. Per fare un ulteriore passo avanti, i due semi-vettori possono a loro volta essere ordinati partizionandoli, ordinando e poi fondendo le loro metà. Così facendo l'equazione di costo assume la forma:

$$\Gamma(N) = 4\Gamma_{seq}\left(\frac{N}{4}\right) + 2\Gamma_{merge}\left(\frac{N}{2}\right) + \Gamma_{merge}(N) = \frac{N^2}{4} + 2N \quad (2.15)$$

Portando il procedimento al limite, il partizionamento può essere ripetuto ricorsivamente fino a operare su vettori elementari di lunghezza 2, che possono essere convenientemente ordinati in tempo costante con un confronto e uno swap. Questo risulta nell'equazione di costo:

$$\Gamma_{MergeS}(N) = \begin{cases} c_1 + \Gamma_{MergeS}\left(\frac{N}{2}\right) + \Gamma_{MergeS}\left(\frac{N}{2}\right) + c_2 \cdot N & \text{se } N > 2 \\ c_3 & \text{se } N == 2 \end{cases} \quad (2.16)$$

dove: c_1 è il costo per decidere se proseguire nel partizionamento; $\Gamma_{MergeS}\left(\frac{N}{2}\right) + \Gamma_{MergeS}\left(\frac{N}{2}\right)$ è il costo per ordinare i due semivettori applicando ricorsivamente il mergesort; $c_2 \cdot N$ è il costo del merge per fondere i due semivettori ordinati; c_3 è il costo del test e dello swap su un vettore di lunghezza 2.

L'approccio generale per pervenire alla soluzione di Eq.(2.18) rientra nei contenuti di un corso di matematica discreta; concettualmente il problema assomiglia ad una equazione differenziale, ma è più semplice perché la formulazione discreta della ricorrenza garantisce che una soluzione esiste ed è unica. Un modo un po' ignorante di trovare la soluzione è quella di sviluppare ripetutamente l'espressione del costo fino ad intuire la forma a cui essa converge:

$$\begin{aligned} \Gamma_{MergeS}(N) &= c_1 + \Gamma_{MergeS}\left(\frac{N}{2}\right) + \Gamma_{MergeS}\left(\frac{N}{2}\right) + c_2 \cdot N \\ &\simeq cN + 2\Gamma_{MergeS}\left(\frac{N}{2}\right) \\ &= cN + 2(c\frac{N}{2} + 2\Gamma_{MergeS}\left(\frac{N}{4}\right)) \\ &= cN + 2(c\frac{N}{2} + 2(c\frac{N}{4} + 2(\dots + 2(\Gamma_{MergeS}\left(\frac{N}{N}\right)))))) \\ &= cN + c2\frac{N}{2} + c4\frac{N}{4} + \dots + cN\frac{N}{N} \\ &= cN\ln(N) \end{aligned} \quad (2.17)$$

Raffinando l'analisi possiamo a questo punto verificare se la forma trovata è effettivamente la soluzione sostitendola nel sistema di Eq.(2.16). Prima di farlo generalizziamo la forma trovata aggiungendo i termini di ordine inferiore a $N\ln(N)$; questo ci porta ad assumere tentativamente la forma:

$$\Gamma_{MergeS}(N) = k_1 + k_2 \cdot N + k_3 \cdot N\ln_2(N) \quad (2.18)$$

Sostituendo in Eq.(2.16) otteniamo:

$$\begin{cases} N\ln_2(N)[k_3 - k_3] + N[k_2 - k_2 + k_3 - c_2] + 1[k_1 - c_1 - 2k_1] = 0 & \forall N > 2 \\ k_1 + 2k_2 + 2k_3 = c_3 \end{cases} \quad (2.19)$$

che a sua volta è soddisfatto se e solo se:

$$\begin{cases} k_1 = -c_1 \\ k_3 = c_2 \\ k_2 = c_3 - 2c_2 + c_1 \end{cases} \quad (2.20)$$

Avendo verificato che la forma di Eq.(2.18) soddisfa il sistema di Eq.(2.16), possiamo concludere che:

$$C_{MergeS} = \Theta(N \ln_2(N)) \quad (2.21)$$

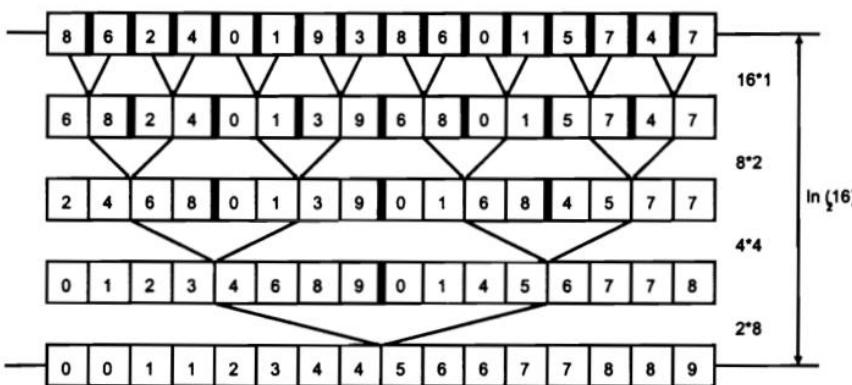


Figura 2.27: L'algoritmo mergesort ha costo $\Theta(N \ln_2(N))$.

In modo più fisico, è possibile arrivare al risultato osservando che su ogni livello di approfondimento della partizione il costo di fusione è pari a N , il che discende dal fatto che la fusione ha costo lineare e il numero di elementi fusi è comunque pari a N su ogni livello (Fig.2.27): nel primo livello si fondono due vettori di lunghezza $\frac{N}{2}$; nel secondo si fondono 4 vettori di lunghezza $\frac{N}{4}$; nel k^{esimo} livello si fondono 2^k vettori di lunghezza $\frac{N}{2^k}$. Dunque il costo è pari a N moltiplicato per il numero di livelli di partizionamento. A sua volta il numero di livelli di partizionamento è pari al numero di volte che è possibile dividere N per 2 prima di ottenere il valore 1, che per definizione è pari al $\ln_2(N)$. Salvo costanti moltiplicative e termini di ordine inferiore questo porta a concludere che il costo dell'algoritmo è pari a $N \cdot \ln_2(N)$.

L'algoritmo ha una implementazione naturale in forma ricorsiva.

Per la ragione dell'efficienza è conveniente evitare di allocare e rilasciare il vettore di appoggio in ciascuna diversa istanza della funzione `merge`: le funzioni `malloc` e `free` richiedono l'intervento del sistema operativo e comportano un costo che può diventare ampiamente dominante rispetto a quello speso in operazioni di confronto e copia. Per ottenere il risultato modifichiamo la definizione

e l'interfaccia della funzione `merge` passando il vettore di appoggio `tmp` tra i parametri.

Per mantenere una interfaccia conforme a quella degli altri algoritmi di ordinamento, conviene poi definire una ulteriore funzione che opera come facciata e nasconde le operazioni di allocazione e rilascio del vettore temporaneo.

```
void _mergesort(struct data * V, int N, struct data * tmp)
/* ordina il vettore V applicando l'algoritmo mergesort.
   il vettore tmp e' riutilizzato nelle diverse istanze per l'appoggio */
{
    if(N>2)
    {
        _mergesort(V,N/2,tmp);
        _mergesort(&V[N/2],N-N/2,&tmp[N/2]);
        merge(V,N,N/2);
    }
    else
    {
        if(V[0]<V[1])
            swap(V,0,1);
    }
}

void mergesort(struct data * V, int N)
/* facciata per la funzione _mergesort.
   Provvede ad allocare e poi rilasciare il vettore di appoggio. */
{
    struct data * tmp;
    tmp=(struct data *)malloc(N*sizeof(struct data));
    _mergesort(V,N,tmp);
    free(tmp);
}
```

2.5.4 Quick-sort

Il quick sort riduce il problema dell'ordinamento al problema della partizione.

Algoritmo di partizione

Il problema della partizione consiste nel riarrangiare gli elementi di un vettore in modo che un elemento pivot sia portato in una posizione tale da avere a sinistra solo elementi di lui minori o uguali (i minoranti) e a destra solo elementi di lui maggiori secchi (i maggioranti) (vedi Fig.2.28). Senza limitazione di generalità viene assunto di prendere come pivot l'elemento che occupa la prima posizione del vettore, condizione alla quale è sempre possibile ricondursi in tempo costante con una operazione di `swap`.

L'algoritmo opera inizialmente sugli elementi dal secondo all' N -esimo partizionandoli in modo che i valori non maggiori del pivot si trovino tutti e soli nelle prime q posizioni a questo punto la condizione di terminazione dell'algoritmo è

raggiunta con uno swap che inverte l'elemento in prima posizione (il pivot) con l'elemento in posizione q . Il valore di q è determinato nel corso dell'esecuzione ed è un risultato restituito dall'algoritmo.

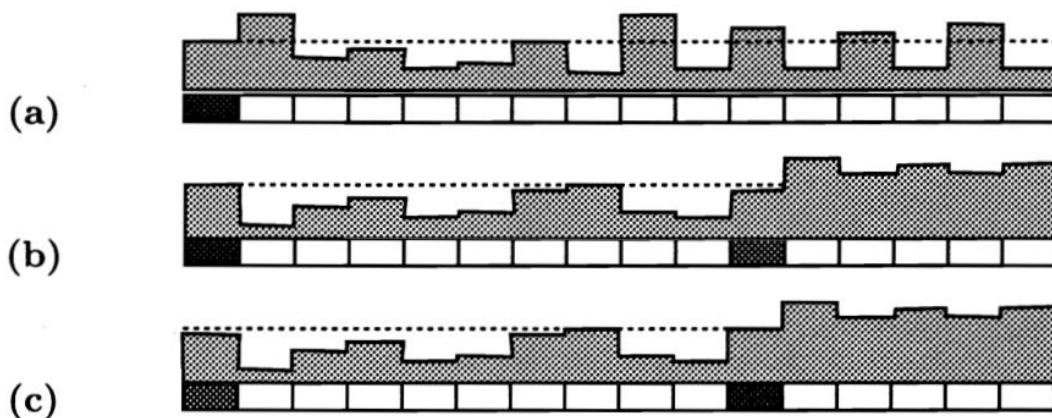


Figura 2.28: L'algoritmo di partizione porta il pivot (il primo elemento nel vettore) nella sua posizione ordinata, e colloca i suoi minoranti alla sua sinistra e i maggioranti alla destra.

La soluzione si basa su due indici l e r che realizzano un invarianto: nel corso della computazione, gli elementi del vettore con indice minore o uguale a l sono minori o uguali al pivot, mentre gli elementi del vettore con indice maggiore o uguale a r sono maggiori del pivot (Fig.2.29).

L'invariante viene inizialmente asserito con $l=0$ e $r=N$, che concretamente indica che il primo elemento è minore o uguale al pivot (in effetti esso stesso è il pivot) mentre nessun elemento è garantito essere maggiore secco del pivot.

Viene a questo punto eseguito uno while-loop che viene ripetuto fino a raggiungere la condizione $l==r-1$. Nel corpo di ciascuna iterazione: prima viene ridotto r fino a che esso indicizza un elemento minore o uguale al pivot; poi viene aumentato l fino a che esso raggiunge r oppure indicizza un elemento maggiore secco del pivot; infine, se l è minore secco di r vengono swappati gli elementi in posizione l ed r . Così facendo, al termine di ciascuna iterazione del while-loop, la distanza tra l ed r si è ridotta ma è stato mantenuto l'invariante per cui gli elementi fino ad l sono minoranti del pivot e quelli da r in poi sono maggioranti del pivot.

Il while-loop termina entro $\frac{N}{2}$ iterazioni. In uscita dal loop, è garantito che: $l==r-1$; gli elementi in posizione da 0 a l sono minori o uguali al pivot; gli elementi nelle posizioni da r a $N-1$ sono maggiori secchi del pivot. L'algoritmo è quindi completato eseguendo uno swap del pivot (i.e. l'elemento in posizione 0) con l'elemento in posizione l . Si noti che così facendo, l'elemento portato in prima posizione nel vettore è comunque un minorante del pivot.

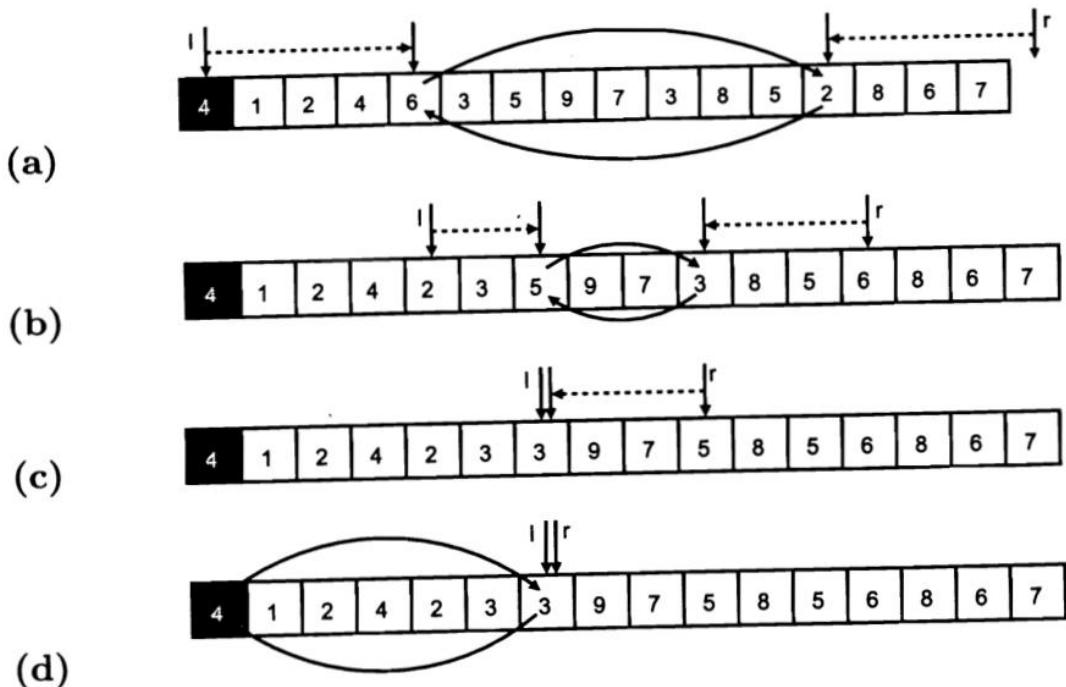


Figura 2.29: L'algoritmo di partizione si basa su due indici l ed r che garantiscono di non avere a sinistra o a destra rispettivamente alcun maggiorante o alcun maggiorante del pivot. Quando i due indici si incontrano, il pivot può essere swappato nella posizione indicata da l raggiungendo la condizione di partizione.

L'algoritmo ha complessità lineare: il numero di swap è al massimo pari al numero di iterazioni del while-loop (il caso si realizza se ad ogni iterazione del while, r decresce di una sola unità e l incrementa di una sola unità); il numero di confronti è proporzionale ad N in quanto ogni elemento del vettore viene comparato contro il pivot una sola volta come parte della iterazione interna che incrementa l oppure (in modo alternativo) come parte della iterazione interna che decrementa r . Dunque $C_{partition} = \Theta(N)$.

La definizione della funzione che implementa l'algoritmo ha alcune sottigliezze sulle quali è opportuno soffermarsi: è rilevante il fatto che le operazioni di riduzione di l e di incremento di r siano operate con cicli do; è rilevante che prima sia ridotto r sotto la sola condizione $V[r] > pivot$ (si noti il $>$) e poi sia aumentato l sotto la doppia condizione che sia $l < r$ e $V[l] \leq pivot$ (si noti il \leq). È anche rilevante osservare come nella ultima iterazione del while può avvenire che r raggiunga l ; in tal caso il ciclo while deve essere terminato re-incrementando r e evitando di incrementare l .

```
int partition(struct data * V, int N)
/* assume il primo elemento come pivot;
```

```

restituisce il numero q di valori in V che sono minori o uguali al pivot,
incluso il pivot stesso;
muove il pivot in posizione q;
mette nelle prime q posizioni tutti e soli i minoranti del pivot;
*/
{
    int l;
    int r;

    pivot=V[0];
    l=0;
    r=N;
    while(l<r)
    {   do{ r--; }while(V[r]>pivot && r>l);
        if(r!=l)
        {   do{ l++; }while(V[l]<=pivot && l<r);
            swap(V,l,r);
        }
    }
    swap(V,l,0);
    return l+1;
}

```

Riduzione dell'ordinamento alla partizione

L'algoritmo di partizione divide un vettore in due sub-vettori separati dal pivot: il pivot è collocato nella posizione che dovrà occupare quando il vettore sarà ordinato; gli elementi del sub-vettore sinistro sono tutti minori o uguali al pivot; gli elementi nel sub-vettore destro sono tutti maggiori secchi del pivot. Questo permette di ordinare separatamente e senza interferenze il sub-vettore dei maggioranti e quello dei minoranti. L'ordinamento dei due sub-vettori viene a sua volta ottenuto ricorrendo all'algoritmo quicksort.

```

void quicksort(struct data * V, int N)
/* quicksort */
{   int q;

    q=partition(V,N);
    quicksort(V,q-1);
    quicksort(&V[q+1],N-q-1);
}

```

Diversamente dal caso del mergesort, il costo dell'algoritmo quicksort dipende dai dati: questi condizionano il modo con cui la partizione divide il vettore e quindi la profondità della ricorsione. In generale, l'equazione di costo è

$$\Gamma_{quicks}(N) = \Gamma_{partition}(N) + \Gamma_{quicks}(q) + \Gamma_{quicks}(N - q - 1) \quad (2.22)$$

dove $\Gamma_{partition}(N) = cN$.

Il caso pessimo si realizza quando q assume sempre un valore estremale (i.e. $q = 0$ oppure $q = N$). In tale condizione, che si realizza ad esempio se il vettore è inizialmente già ordinato o se esso è esattamente in controordine, l'equazione diventa:

$$\Gamma_{quicks}(N) = cN + \Gamma_{quicks}(0) + \Gamma_{quicks}(N - 1) = cN + \Gamma_{quicks}(N - 1) \quad (2.23)$$

Questa ha la stessa forma del selection sort: spendendo N per una partizione si riduce di 1 la dimensione dei dati da trattare. Evidentemente questo risulta in un costo del tipo $\Gamma_{quicks}(N) = c\frac{N(N-1)}{2}$ che conduce a una complessità quadratica:

$$C_{quicks} = \Theta(N^2) \quad (2.24)$$

Pur ottenendo la medesima complessità del più semplice selection sort, il quicksort è un algoritmo efficiente. Questo dipende dal fatto che il suo costo medio differisce dal costo del caso pessimo e assume invece la forma di $N \ln_2(N)$. Si osservi che tra i vari algoritmi trattati, questo è l'unico nel quale il costo medio ha una forma diversa da quella del costo pessimo.

Per derivare il costo medio passiamo al valor medio sull'uguaglianza di Eq.(2.22) marginalizzando¹ sul valore q del numero dei minoranti restituito dalla partizione:

$$\bar{\Gamma}_{quicks}(N) = cN + \sum_{q=0}^{N-1} P_q \cdot \bar{\Gamma}_{quicks}(q) + \sum_{q=0}^{N-1} P_q \cdot \bar{\Gamma}_{quicks}(N - q - 1) \quad (2.25)$$

P_q denota la probabilità che q sia il numero dei minoranti restituito dall'algoritmo di partizione, ovvero esso denota la probabilità che q sia la posizione del pivot nel vettore ordinato. Se assumiamo che i valori nel vettore siano randomizzati, ovvero che la loro distribuzione non sia condizionata da un qualche fattore di ordinamento, possiamo assumere che $P_q = \frac{1}{N}$ indipendentemente dal valore di q . Si noti inoltre che le due sommatorie in Eq.(2.25) sono uguali, sommando gli stessi termini in diverso ordine. Possiamo allora riscrivere $\bar{\Gamma}_{quicks}(N)$ nella forma:

$$\bar{\Gamma}_{quicks}(N) = cN + \frac{2}{N} \sum_{q=0}^{N-1} \bar{\Gamma}_{quicks}(q) \quad (2.26)$$

¹Se X è funzione della variabile aleatoria q la quale assume i valori q_i con $i = 0, N - 1$ con probabilità p_i , il valore medio di X può essere espresso nella forma $\bar{X} = \sum_{i=0}^{N-1} \bar{X}(q_i) \cdot p_i$.

Eq(2.25) ci permette ora di dimostrare che esistono due costanti C e D tali che:

$$\bar{\Gamma}_{quicks}(k) \leq C(k+1)\ln(k+1) + D \quad (2.27)$$

La dimostrazione procede per induzione sulla dimensione del vettore a cui è applicata la partizione. Per ipotesi induttiva, supponiamo che la disequazione di Eq.(2.27) sia valida per ogni $k < N$ e dimostriamo che essa vale anche per N .

$$\begin{aligned} \bar{\Gamma}_{quicks}(N) &= cN + \frac{2}{N} \sum_{q=0}^{N-1} \bar{\Gamma}_{quicks}(q) \leq cN + 2D + \frac{2C}{N} \sum_{q=1}^N q \ln(q) \leq \\ &cN + 2D + \frac{2C}{N} \left(\sum_{q=1}^{N/2} q \ln(q) + \sum_{q=N/2+1}^N q \ln(q) \right) \leq \\ &cN + 2D + \frac{2C}{N} \left(\ln(N/2) \sum_{q=1}^{N/2} q + \ln(N) \sum_{q=N/2+1}^N q \right) = \\ &cN + 2D + \frac{2C}{N} \left((\ln(N) - \ln(2)) \sum_{q=1}^{N/2} q + \ln(N) \sum_{q=N/2+1}^N q \right) = \\ &cN + 2D + \frac{2C}{N} \left(\ln(N) \sum_{q=1}^N q - \sum_{q=1}^{N/2} q \right) = \\ &cN + 2D + \frac{2C}{N} \left(\ln(N) \frac{N(N-1)}{2} - \frac{N/2(N/2-1)}{2} \right) \leq \\ &cN + 2D + CN \ln(N) - \frac{CN}{4} \leq CN \ln(N) + D \end{aligned} \quad (2.28)$$

dove l'ultimo passaggio vale per qualunque scelta delle costanti C e D che da qualche valore di N in poi soddisfino la condizione:

$$cN + 2D - \frac{CN}{4} \leq 0 \quad (2.29)$$

Nella derivazione di Eq.(2.28) si osservi che la sommatoria $\sum_{q=1}^N q \ln(q)$ è stata scomposta in due somme da 0 a $N/2$ e da $N/2+1$ a N . Questo nella stima finale fa la differenza di un fattore 2 al denominatore che è essenziale per compensare il fattore 2 che moltiplica il coefficiente C .

È interessante osservare che per l'algoritmo quicksort il costo medio coincide con il costo nel caso ottimo. Questo si realizza quando il numero di minoranti q restituito dalla partizione assume sempre la posizione mediana: $q = \lceil \frac{N}{2} \rceil$. In tale condizione il vettore viene sempre ripartito in due sub-vettori di uguale dimensione

minimizzando così la profondità di ricorsione. Applicando l'ipotesi di ottimalità sulla forma generale del costo di Eq.(2.22):

$$\begin{aligned}\Gamma_{quicks}(N) &= \Gamma_{partition}^{best}(N) + \Gamma_{quicks}^{best}(\lceil \frac{N}{2} \rceil) + \Gamma_{quicks}^{best}(\lceil \frac{N}{2} \rceil) \\ &= cN + 2 * \Gamma_{quicks}^{best}(\lceil \frac{N}{2} \rceil)\end{aligned}\quad (2.30)$$

Questa è la stessa forma del costo del mergesort e conduce a una soluzione della forma

$$\Gamma_{quicks}^{best}(N) = N \ln_2(N) \quad (2.31)$$

È interessante soffermarsi a riflettere su come il costo medio assuma la forma del caso ottimo e non quella del caso pessimo, cosa che invece avviene per tutti gli altri algoritmi fino a qui considerati. Il fatto che il caso medio assuma la forma del caso ottimo *non* dipende in alcun modo da una qualche asimmetria nella probabilità di estrarre un pivot centrale che ripartisce bene il vettore rispetto alla probabilità di estrarre un pivot estremale che non ripartisce: come abbiamo anche osservato nella derivazione del costo medio, la probabilità che il pivot assuma una qualsiasi delle N posizioni disponibili è uniforme. La vera ragione del fenomeno consiste nel fatto che l'impatto prodotto sul processo di ripartizione da un pivot buono prevale sull'impatto che produce un pivot cattivo. Ragionando in modo molto ruvido, immaginiamo che i pivot siano estratti con sorti alternate: una volta estraiamo un pivot estremale e una volta quello centrale. In questa condizione il costo assume la forma $2 \cdot N \ln(N)$ dove il fattore 2 dà conto del fatto che una volta sua due non abbiamo progresso nella riduzione del numero dei dati da trattare.

È ragionevole a questo punto chiedersi se possa essere conveniente condizionare la selezione del pivot in modo da aumentare la probabilità che esso assuma una posizione mediana nel vettore.

Prima di attaccare il problema è conveniente stabilire quale è il limite di complessità che possiamo permetterci nella selezione del pivot. Se il pivot viene selezionato in tempo lineare, la complessità dell'algoritmo di partizione mantiene la forma $\Gamma_{partition}(N) = c'N$ e quindi non cambiano le derivazioni successive. Viceversa se la selezione del pivot richiede un tempo $N \ln(N)$ è possibile verificare (in modo non del tutto banale), che la derivazione del costo nel caso ottimo conduce ad una complessità che eccede la forma $N \ln(N)$. Non essendovi altre forme intermedie tra $\Theta(N)$ e $\Theta(N \ln(N))$, possiamo concludere che un eventuale algoritmo di selezione del pivot deve comunque operare con complessità non superiore a $\Theta(N)$.

Purtroppo un algoritmo che seleziona il mediano in tempo lineare non esiste. Anzi, nell'approccio più comune, il problema di selezione del mediano di un vettore viene proprio ridotto all'ordinamento: prima viene ordinato il vettore (con

complessità $N \ln(N)$) e poi viene selezionato l'elemento in posizione centrale (in tempo costante).

2.5.5 Heap-sort

L'algoritmo heapsort combina i pregi del mergesort e del quicksort: opera in tempo $\Theta(N \ln_2(n))$ anche nel caso pessimo e non richiede un'area di appoggio di dimensione dipendente dall'insieme dei dati.

I dati sono memorizzati su un *albero binario completo* rappresentato in *forma sequenziale*: i valori sono collocati su un vettore di dimensione M maggiore o uguale alla dimensione dell'insieme dei dati (i.e. $M \geq N$); la radice è memorizzata nella locazione 0 del vettore; i successori sinistro e destro del nodo in posizione k sono memorizzati nelle posizioni $left(k) = 2k + 1$ e $right(k) = 2k + 2$ (cfr. Capitolo 2.2.3).

Condizione di heap

L'algoritmo si basa sulla capacità di arrangiare i dati nell'albero in modo da soddisfare una particolare condizione di ordinamento detta *condizione di heap*: il valore su un qualsiasi nodo k è maggiore o uguale dei valori contenuti nei due eventuali figli $left(k)$ e $right(k)$ (Fig. 2.30). Come corollario, questo implica che il valore nel nodo k è maggiore o uguale di qualunque valore contenuto nei sottoalberi $T_{left(k)}$ e $T_{right(k)}$ che hanno radice nei figli $left(k)$ e $right(k)$.

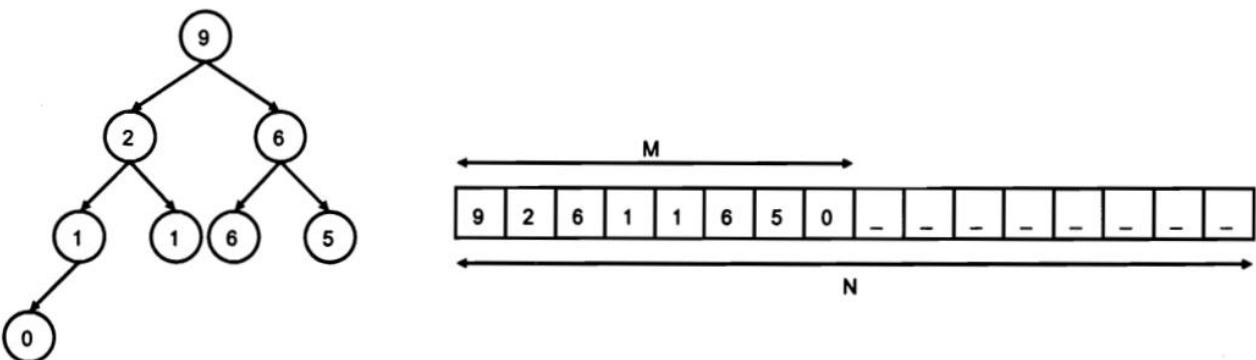


Figura 2.30: Un albero binario completo che soddisfa la condizione di heap e la sua rappresentazione in forma sequenziale.

La asserzione iniziale e il mantenimento della condizione di heap è a sua volta fondata su un'operazione, che chiameremo `propagate_heap`, la quale asserisce la condizione di heap sull'albero con radice in locazione k assumendo che i due sottoalberi di k soddisfino già la condizione.

Se k non ha un figlio che gli sia maggiore, la condizione di heap è già soddisfatta. Altrimenti k viene swappato nella posizione del maggiore tra i due figli. Il sottoalbero non interessato dallo swap e la nuova radice in posizione k soddisfano la condizione di heap, mentre il sottoalbero che ha subito lo swap potrebbe non soddisfare più la condizione di heap ma ha a sua volta due sottoalberi che soddisfano la condizione. L'operazione può quindi essere conclusa applicando ricorsivamente l'operazione di `propagate_heap` al sotto-albero perturbato (Fig.2.31). Il codice c è forse più chiaro delle parole:

```

void propagate_heap(float * V, int k, int N)
/* asserisce la condizione di heap sull'albero con radice in k
   assumendo che la condizione sia soddisfatta sui due sottoalberi di k.
   Assume la responsabilità di distinguere se k ha entrambi i figli,
   se ha solo il figlio sinistro, o se è una foglia */
{
    if(right(k)<N)                                // k ha due figli
    {
        if(V[left(k)]>V[right(k)])
        {   if(V[k] >= V[left(k)])
            return;
            else
                propagate_heap(V,left(k),N);
        }
        else
        {   if(V[k] >= V[right(k)])
            return;
            else
                propagate_heap(V,right(k),N);
        }
    }
    else if(left(k)<N)                            // k ha il solo figlio sinistro
    {   if(V[k] >= V[left(k)])
        return;
        else
            propagate_heap(V,left(k),N);
    }
    else                                            // k è una foglia
        return;
}

```

Il costo dell'algoritmo `propagate_heap` dipende dalla distanza dalle foglie del nodo a cui è applicata l'operazione, ovvero dalla differenza tra la profondità D dell'albero e la profondità d del nodo:

$$\Gamma_{\text{propagate_heap}}(D, d) = \begin{cases} c_1 + \Gamma_{\text{propagate_heap}}(d+1) & \text{per } d < D \\ c_2 & \text{per } d == D \end{cases} \quad (2.32)$$

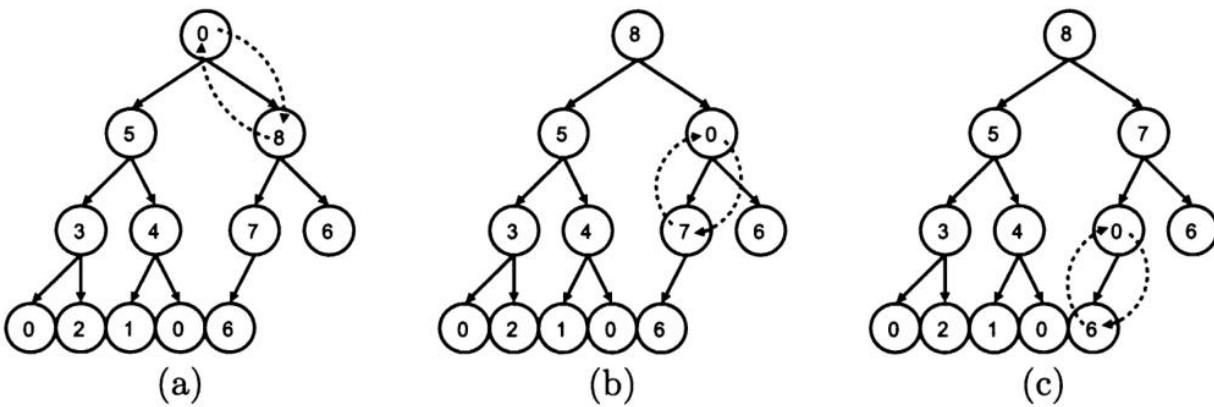


Figura 2.31: (a) Un albero binario completo nel quale i due sottoalberi della radice soddisfano la condizione di heap. (a-b-c) sequenza di swap che estende alla radice la condizione di heap attraverso l'operazione `propagate_heap`.

da cui segue

$$\Gamma_{\text{propagate_heap}}(D, d) = c(D - d)$$

La condizione di heap può essere stabilita su tutti i nodi di un albero con una operazione che chiameremo `set_heap`. Questa applica ripetutivamente l'operazione `propagate_heap` su tutti i nodi, procedendo dalle foglie verso la radice. Così facendo inizialmente sono tratte le foglie (che soddisfano la condizione per definizione) e la condizione viene stabilita su un nodo k solo dopo averla stabilita nella sua discendenza. Si osservi che, per la particolare disposizione dei nodi su un albero binario in forma sequenziale, la visita a partire dalle foglie si ottiene con una semplice scansione sequenziale sul vettore di rappresentazione.

```

void set_heap(float * V, int N)
/* assicura la condizione di heap sull'intero albero;
   procede dalle foglie verso la radice; */
{
    int k;

    for(k=N-1; k>=0; k--)
        propagate_heap(V,k,N);
}

```

Il costo di `set_heap` è la somma del costo complessivo per la applicazione di `propagate_heap` a tutti i nodi dell'albero. Tenendo conto che sul livello d ci sono 2^d nodi, questo può essere espresso nella forma:

$$\Gamma_{set_heap}(D) = \sum_{d=0}^D 2^d \Gamma_{propagate_heap}(D, d) = c2^D \sum_{d=0}^D \frac{D-d}{2^{D-d}} = c2^D \sum_{d=0}^D \frac{d}{2^d} \quad (2.33)$$

da cui segue:

$$\Gamma_{set_heap}(D) \leq c2^D \quad (2.34)$$

Riduzione dell'ordinamento all'estensione dello heap e allo swap

L'algoritmo heapsort opera su un albero di M elementi che soddisfa la condizione di heap codificato in forma sequenziale. Il massimo si trova sulla radice dell'albero e quindi in posizione 0 nel vettore (Fig.2.32).

Effettuando uno swap tra le posizioni 0 e $M - 1$ nel vettore, il massimo viene collocato nella sua posizione ordinata. Inoltre, i primi $M - 1$ valori del vettore sono organizzati nella forma di albero binario sequenziale nel quale i due sottoalberi della radice soddisfano la condizione di heap. Applicando l'operazione `propagate_heap` sulla radice si ripristina la condizione di heap sull'intero albero. Questo riconduce alla condizione iniziale avendo però ridotto il problema dalla dimensione M a $M - 1$.

L'algoritmo completa l'ordinamento ripetendo $M - 1$ lo swap tra la radice dell'albero e l'ultima posizione del vettore e ripristinando ogni volta la condizione di heap con una chiamata alla funzione `propagate_heap` a partire dalla radice:

```
void heap_sort(float * V, int N)
/* ordinamento heap sort; */
{
    set_heap(V,N);

    for(n=N-1;n>=0;n--)
    {
        swap(V,0,n)
        propagate_heap(V,0,n);
    }
}
```

Il costo complessivo dell'ordinamento heapsort è pari al costo per il `set_heap` iniziale più quello della successiva iterazione, nella quale si eseguono $N - 1$ swap e `propagate_heap` su un albero di dimensione decrescente, sempre a partire dalla radice:

$$\begin{aligned} \Gamma_{Heap_sort}(N) &= \Gamma_{set_heap}(N) + \sum_{n=N-1}^1 \Gamma_{propagate_heap}(\lfloor \ln_2(n), 0 \rfloor) = \\ &= c_1 N + c_2 \sum_{n=N-1}^1 \lfloor \ln_2(n) \rfloor \geq c_1 N + c_2 \sum_{n=N-1}^1 \ln_2(N) = c_1 N + c_2 N \ln_2(N) \end{aligned} \quad (2.35)$$

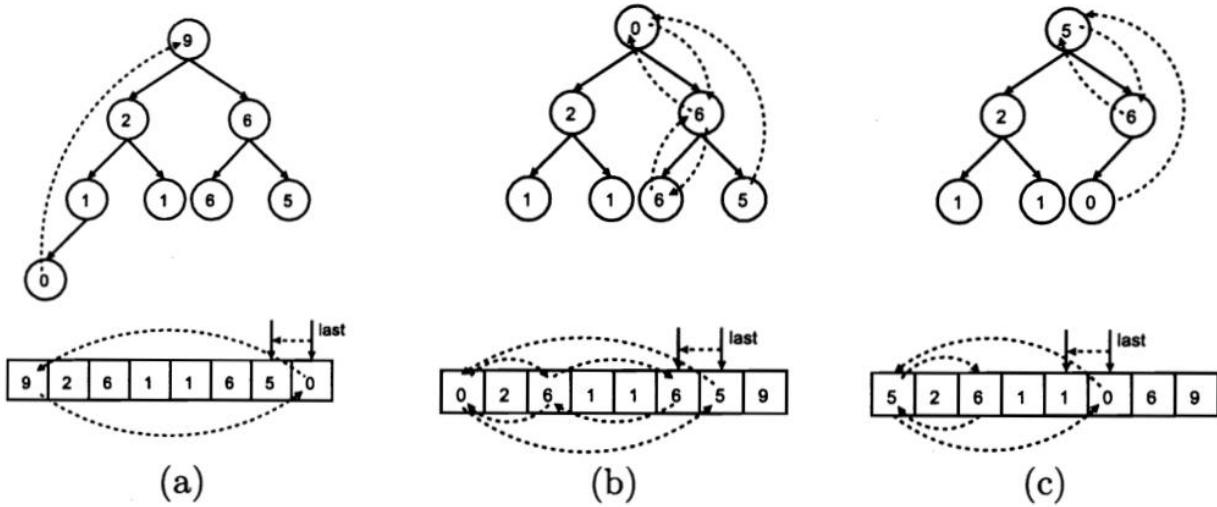


Figura 2.32: L'algoritmo heapsort. (a) I dati sono inizialmente codificati su un albero binario in forma sequenziale che soddisfa la condizione di heap. (a-b-c) Ripetutamente, la radice viene estratta dall'albero e swappata con il valore che occupa la sua posizione ordinata (linee a tratteggio) e la condizione di heap viene poi ripristinata applicando l'operazione di `propagate_heap` a partire dalla radice (linee punteggiate). Dopo k passi, i k maggiori valori del vettore sono in ordine.

da cui:

$$C_{\text{Heap-Sort}} = \Theta(N \ln_2(N))$$

Quicksort, mergesort e heapsort

Gli algoritmi MergeSort e QuickSort hanno alcune differenze e similitudini che è conveniente osservare.

Entrambi gli algoritmi applicano un meccanismo di partitionamento per trattare separatamente sotto-insiemi dei dati. Poiché il problema ha complessità di ordine superiore a quello lineare, questo risulta in una riduzione di complessità. In questo il mergesort è più efficiente ottenendo sempre una ripartizione ottimale, che garantisce di risolvere il problema con complessità $\Theta(N \ln_2(N))$ indipendentemente dal valore dei dati. Viceversa, nel quicksort la partizione dipende dai dati e può degenerare in modo da fornire una complessità $\Theta(N^2)$. Il mergesort ha tuttavia il maggiore limite di richiedere un vettore di appoggio di dimensione proporzionale all'insieme dei dati.

Entrambi gli algoritmi risolvono il problema dell'ordinamento limitandosi a coordinare le chiamate ad un algoritmo di complessità lineare (merge o partition nei due casi). Le implementazioni ricorsive che abbiamo illustrato evidenziano questa organizzazione: in entrambi i casi, l'algoritmo è fatto di una guardia sul-

la dimensione del vettore e di un corpo costituito di due istruzioni di chiamata ricorsiva e una istruzione di chiamata al sub-algoritmo di complessità lineare.

Mentre mergesort opera in forma positicipata collocando le chiamate ricorsive prima della operazione di fusione, il quicksort opera in forma anticipata operando la partizione prima di distribuire il controllo ricorsivamente (si noti che la differenza è catturata nella forma con cui sono espresse le leggi di costo dei due algoritmi in Eq.(2.16) e Eq.(2.22)). Sviluppando nel tempo la sequenza delle chiamate, ne risulta che il mergesort lavora dal basso effettuando degli swap solo dopo avere distribuito il controllo fino al massimo livello di suddivisione del vettore, mentre il quicksort lavora dall'alto effettuando le operazioni di swap prima di ripartire il vettore.

Lo heap sort migliora sia il quicksort che il mergesort, combinando una complessità $\Theta(N \ln_2(N))$ indipendentemente dai dati con la capacità di operare in place senza richiedere una memoria di appoggio di dimensione non costante.

È interessante osservare che lo heapsort riduce la complessità al tempo $\Theta(N \ln_2(N))$ non attraverso la partizione dei dati ma piuttosto mantenendo i dati nella forma di heap. Questa può essere riguardata come una forma intermedia tra l'avere il vettore in arrangiamento casuale e averlo in forma ordinata. Ha il prezzo di una forma che raggiunge l'equilibrio necessario per permettere la selezione del massimo in tempo costante e per potere essere ripristinata in tempo logaritmico dopo la perturbazione indotta dall'estrazione del massimo. Diversamente da quicksort e mergesort, questo conduce all'ordinamento attraverso un numero lineare di chiamate a operazioni di complessità logaritmica.

In questa prospettiva, lo heapsort può essere riguardato come una ottimizzazione del selection sort, del quale ripete il meccanismo di progressiva riduzione dell'insieme dei dati attraverso la selezione e il posizionamento del massimo.

2.6 Complessità minima di un problema

I diversi algoritmi che abbiamo considerato per risolvere il problema dell'ordinamento hanno tutti una complessità non minore di $N \ln_2(N)$. È possibile pensare ad un algoritmo che faccia di meglio?

La formulazione generale di una questione di questo tipo chiama in gioco il concetto di *complessità minima di un problema*: si dice che un problema prob ha complessità minima $f(n)$, e si scrive $C_{\text{prob}} = \Omega(f(n))$, se per qualsiasi algoritmo Alg che risolva prob risulta $C_{\text{Alg}} \geq \Theta(f(n))$ (dove la relazione di ordine non stretto \geq tra ordini di grandezza ha il significato già definito nella sezione 2.3.2). Concretamente, la complessità minima di un problema costituisce una stima dal basso del massimo del costo che può essere speso nel risolvere un problema al variare dell'algoritmo e dei dati di ingresso.

In generale, la valutazione della complessità minima di un problema fornisce un risultato negativo che ci risparmia dal ricercare soluzioni o miglioramenti dove non ne possono esistere. Vedendo il bicchiere mezzo pieno, la complessità minima di un problema ci abilita a qualificare la *ottimalità* di un algoritmo: un algoritmo si dice ottimale se la sua complessità (massima) uguaglia la complessità (minima) del problema che esso risolve.

Tornando al caso specifico del problema dell'ordinamento, con un ragionamento abbastanza ruvido è facile convincersi che algoritmi di complessità $\Theta(N \ln_2(N))$, quale possono essere il mergesort o lo heapsort, costituiscono una soluzione ottima. In effetti, per ridurre la complessità ad un ordine inferiore a $N \ln_2(N)$ dovremmo trovare un algoritmo che risolvesse l'ordinamento in tempo N , visto che tra $\Theta(N)$ e $\Theta(N \ln_2(N))$ non esistono ordini di grandezza intermedi e distinti. E d'altra parte si intuisce che non sia possibile ordinare N valori in tempo $\Theta(N)$: questo significherebbe trattare ciascun dato un numero costante di volte (indipendente da N), che non sembra compatibile col fatto che l'ordinamento dipende non dai valori individuali dei singoli dati ma piuttosto dalla loro relazione.

Oltre questa prima intuizione, è possibile fornire una dimostrazione formale. Che non solo ha il valore di chiudere quantitativamente la trattazione dei diversi algoritmi di ordinamento, ma che anche esemplifica concretamente un modo di affrontare la derivazione di un risultato che attiene non ad uno specifico algoritmo ma piuttosto alla infinita varietà di algoritmi che possono risolvere un problema.

2.6.1 La complessità del problema dell'ordinamento

Il problema dell'ordinamento consiste nel trovare una permutazione delle posizioni degli elementi di un vettore che soddisfi la condizione di ordinamento. Per dimostrare che il problema non può essere risolto con alcun algoritmo di complessità

inferiore a $\Theta(N \ln_2(N))$, supponiamo di avere un algoritmo Alg che ordina un vettore V di dimensione N e dimostriamo che è possibile assegnare agli elementi del vettore V valori che lasciano indeterminata la permutazione giusta almeno fino a che Alg ha effettuato $N \ln_2(N)$ confronti. La strategia della dimostrazione consiste nell'evitare di dichiarare inizialmente il valore degli elementi di V e definirli invece in base alla sequenza delle condizioni determinate nei confronti effettuati nell'esecuzione di Alg .

Denotiamo con S^k l'insieme delle permutazioni $Perm$ per cui esiste assegnamento dei valori di V che è compatibile con i primi k confronti effettuati nell'esecuzione di Alg e che fa sì che $Perm$ risulti essere la soluzione del problema di ordinamento.

Inizialmente S^0 è costituito dall'insieme di tutte le $N!$ possibili permutazioni degli elementi del vettore ²:

$$||S^0|| = N! \quad (2.36)$$

Infatti, per una qualsiasi permutazione $Perm$, esistono infiniti valori che possono essere assegnati agli elementi di V in modo che $Perm$ sia la permutazione ordinata. Nel modo più semplice questo può essere ottenuto attribuendo a V gli stessi valori di $Perm$: se ad esempio assegniamo al vettore V i valori $V = [2, 1, 0, 4]$, la permutazione ordinata diventa $Perm = [2, 1, 0, 4]$.

Gli esiti dei confronti effettuati nell'esecuzione dell'algoritmo pongono dei vincoli sui valori che possono essere attribuiti agli elementi del vettore V , riducendo così l'insieme delle permutazioni che hanno la potenzialità di essere quella ordinata. In questa prospettiva è interessante osservare che, in ultimo, la differenza tra i diversi algoritmi di ordinamento che abbiamo considerati consiste nella strategia con cui sono sequenzializzati i confronti tra i diversi elementi del vettore e quindi nella rapidità con cui i diversi confronti riducono l'insieme delle permutazioni accettabili.

Quantitativamente, la determinazione del $k + 1$ -esimo confronto divide S^k in due parti S_t e S_f che contengono rispettivamente tutte e sole le permutazioni in S^k che sono compatibili con le due condizioni alternative in cui il $k + 1$ -esimo confronto ha esito **true** e **false**, rispettivamente.

Poiché la somma delle dimensioni di S_f e S_t è pari alla dimensione di S^k (i.e. $||S_t|| + ||S_f|| = ||S^k||$), uno dei due sotto-insiemi, che denotiamo come S^* , contiene almeno la metà degli elementi di S^k (i.e. $||S^*|| = \text{Max}\{||S_t||, ||S_f||\} \geq$

²Dovendo disporre i numeri naturali compresi tra 0 e $N - 1$ in N posizioni, il valore 0 può occupare una qualsiasi di N locazioni, il valore 1 una qualsiasi delle $N - 1$ che non sono occupate dallo 0, il valore 2 in una qualsiasi delle $N - 2$ locazioni che non sono occupate né dallo 0 né dall'1, e così via. Questo risulta in ultimo in $N \cdot (N - 1) \cdot (N - 2) \cdot \dots \cdot 1 = \prod_{n=1}^N = N!$ configurazioni che sono distinte e coprono tutte le diverse possibili permutazioni.

$\frac{1}{2}||S^k||$). Poiché per costruzione S^* è non vuoto, possiamo assumere che il $k+1$ -esimo confronto abbia l'esito per cui S^* diventa l'insieme delle permutazioni possibili dopo $k+1$ confronti (i.e. $||S^{k+1}| = S^*||$). Risulta allora:

$$||S^{k+1}|| \geq \frac{1}{2}||S^k|| \quad (2.37)$$

L'algoritmo non può terminare prima che l'insieme delle permutazioni accettabili abbia dimensione 1. Infatti, se l'algoritmo esprimesse un tentativo prima di quel momento, sarebbe sempre possibile una scelta dei valori di V che conducesse ad una permutazione ordinata compatibile con tutti i vincoli espressi nei confronti e tuttavia diversa dal tentativo espresso. Il numero minimo di confronti k_{min} che l'algoritmo deve effettuare prima di avere le condizioni necessarie per terminare è quindi caratterizzato dall'equazione:

$$||S^{k_{min}}|| = 1 \quad (2.38)$$

Componendo le equazioni (2.36), (2.37) e (2.38), e indicando con d^k la cardinalità $||S^k||$, si ottiene il sistema:

$$\begin{cases} d^0 = N! \\ d^{k+1} \geq \frac{1}{2}d^k \\ d^{k_{min}} = 1 \end{cases} \quad (2.39)$$

da cui segue che k_{min} è non minore del numero di volte che è possibile dividere per 2 il valore $N!$ prima di ottenere l'unità. Ovvero, per la stessa definizione di logaritmo binario:

$$k_{min} = \ln_2(N!) \quad (2.40)$$

La derivazione è conclusa applicando la formula di Stirling ($N! \simeq (\frac{N}{e})^N$):

$$k_{min} = \ln_2(N!) = \ln_2((\frac{N}{e})^N) = N \ln_2(N) - N \ln_2(e) \geq N \ln_2(N) \quad (2.41)$$

2.7 Esercizi

Es. 2.7.1

Si definisca la funzione che verifica se due liste di interi rappresentate in forma collegata con puntatori sono uguali.

Soluzione: Due liste sono uguali se hanno la stessa lunghezza e gli stessi valori.

```
struct list{
    int value;
    struct list * next_ptr;
};

Boolean is_equal(struct list * ptr1, struct list * ptr2)
/* Verifica se le liste ptr1 e ptr2
hanno la stessa lunghezza e gli stessi valori. */
{
    Boolean mismatch_found;

    mismatch_found=FALSE;
    while(ptr1!=NULL && ptr2!=NULL && mismatch_found==FALSE)
    {
        if(ptr1->value!=ptr2->value)
            mismatch_found=TRUE;
        else
        {
            ptr1=ptr1->next_ptr;
            ptr2=ptr2->next_ptr;
        }
    }
    if(ptr1==NULL && ptr2==NULL && mismatch_found==FALSE)
        return TRUE;
    else
        return FALSE;
}
```

□

Es. 2.7.2 *Si definiscano due funzioni c che calcolano la lunghezza di una lista di interi, in forma iterativa e in forma ricorsiva.*

Soluzione: Nella versione iterativa, la lunghezza viene calcolata visitando la lista e incrementando un accumulatore a ogni elemento visitato. Nella versione ricorsiva, la lista vuota ha lunghezza 0, mentre la lista non vuota ha lunghezza 1 più la lunghezza della lista suffissa al primo elemento.

```
int IIlength (struct list * ptr)
/* Calcola la lunghezza di una lista. Implementazione iterativa. */
{   int length;
```

```

length=0;
while(ptr!=NULL)
{   length++;
    ptr=ptr->next_ptr;
}
return length;
}

float Rlength (struct list *ptr)
/* Calcola la lunghezza di una lista. Implementazione ricorsiva. */
{
    if(ptr!=NULL)
        return 1+Rlength(ptr->next_ptr);
    else
        return 0;
}

```

□

Es. 2.7.3 In riferimento all'esercizio 2.7.2, si stimi la quantità di memoria impegnata dalle due implementazioni in forma iterativa e ricorsiva.

Soluzione: La versione iterativa richiede l'allocazione del parametro `ptr` (un indirizzo) e della variabile locale `length` (un intero). La versione ricorsiva non alloca variabili ma esercita un carico sullo stack lineare rispetto alla lunghezza della lista: per ogni chiamata viene memorizzato sullo stack il parametro `ptr`.

□

Es. 2.7.4 Si definisca la funzione `c` in forma ricorsiva che crea la copia di una lista collegata con puntatori.

Soluzione: Possiamo ripartire l'implementazione tra una facciata che esegue le operazioni di inizializzazione (della lista destinazione) e una funzione ricorsiva che ripetutamente esegue l'operazione di inserimento. Nella ripetizione assumiamo uno schema posticipato (i.e. l'invocazione ricorsiva precede l'inserimento) che risulta in una visita in contro-ordine della lista sorgente e permette di eseguire gli inserimenti in testa. Così facendo, gli inserimenti hanno costo costante e l'intera copia è eseguita in tempo lineare.

```

void clone(struct list * src_ptr, struct list ** dst_ptrptr)
{
    init(dst_ptrptr);
    clone_initialized(src_ptr,dst_ptrptr);
}

```

```

void clone_initialized(struct list * src_ptr, struct list ** dst_ptrptr)
{
    if(src_ptr!=NULL)
    {
        clone_initialized(src_ptr->next_ptr,dst_ptrptr);
        pre_insert(src_ptr->value,dst_ptrptr);
    }
}

```

L'implementazione può essere resa più compatta senza alterare la complessità computazionale eliminando la facciata e attribuendo l'operazione di inizializzazione alla clausola che tratta il caso terminale della ricorsione:

```

void clone2(struct list * src_ptr, struct list ** dst_ptrptr)
{
    if(src_ptr!=NULL)
    {
        clone2(src_ptr->next_ptr,dst_ptrptr);
        pre_insert(src_ptr->value,dst_ptrptr);
    }else
        init(dst_ptrptr);
}

```

□

Es. 2.7.5 Si definisca la funzione c in forma ricorsiva che cancella una lista collegata con puntatori rilasciando la memoria.

Soluzione: Poiché gli elementi della lista devono essere cancellati a partire dall'ultimo usiamo uno schema posticipato (i.e. l'invocazione ricorsiva precede l'operazione di cancellazione). Si osservi che, diversamente da quanto avviene nell'esercizio 2.7.4 dove esisteva un'operazione di inizializzazione, in questo caso esiste un'operazione di chiusura (ripristinare il valore null sul puntatore di testa della lista), la quale non può essere attribuita alla clausola che tratta il caso terminale della ricorsione e deve invece essere eseguita in una funzione di facciata.

```

void delete_list(struct list ** ptrptr)
{ delete(ptrptr);
  *ptrptr=NULL
}

void delete(struct list ** ptrptr)
{
    if(*ptrptr!=NULL)
    { delete(&(*ptrptr)->next_ptr));
      free(*ptrptr);
    }
}

```

□

Es. 2.7.6 Si definisca la funzione *c* in forma iterativa che cancella una lista collegata con puntatori rilasciando la memoria.

Soluzione: L'implementazione in forma iterativa è resa difficile dalla impossibilità di scandire la lista in ordine inverso per cancellare gli elementi a partire dall'ultimo.

A prima vista il problema potrebbe essere risolto seguendo un approccio generale con cui con cui è possibile tradurre in forma iterativa una forma ricorsiva: con una visita della lista sono caricati su uno stack applicativo i puntatori agli elementi della lista stessa; successivamente gli indirizzi sullo stack sono prelevati (in ordine LIFO) per eseguire i rilasci in ordine dall'ultimo al primo. Tuttavia, non essendo a priori nota la dimensione della lista, lo stack stesso dovrebbe essere implementato come lista (con operazioni di inserimento e cancellazione in testa). Al termine della cancellazione della lista originaria, rimarrebbe quindi da cancellare la lista creata per appoggiarvi i puntatori!

Conviene allora rinunciare a cancellare gli elementi a partire dall'ultimo, effettuando cancellazioni in testa fino a che la testa punta un elemento nullo:

```
void delete_list_i(struct list ** ptrptr)
{ struct list * tmp_ptr;

    while(*ptrptr!=NULL)
    { tmp_ptr=*ptrptr;
        ptrptr=&((*ptrptr)->next_ptr);
        free(tmp_ptr);
    }
}
```

□

Es. 2.7.7 Si valuti la complessità della soluzione proposta per l'esercizio 1.6.7.

Soluzione: nel caso pessimo, per ogni valore di base da 0 fino a $N_1 - N_2$, viene variato offset da 0 a N_2 . Per ogni passo della scansione viene effettuato un confronto che ha costo costante. La componente dominante del costo totale è quindi della forma $N_1 \cdot N_2$ e la complessità è $\Theta(N_1 \cdot N_2)$ □

Es. 2.7.8 Si definisca la funzione *c* che verifica se una lista in forma collegata con puntatori contiene due valori uguali consecutivi.

Soluzione: La soluzione riproduce sostanzialmente lo schema di una ricerca. Come unica differenza, la condizione di legalità richiede che siano non nulli sia il puntatore all'elemento corrente che quello al suo successore:

```
Boolean check_pairs(struct list * ptr)
{ Boolean pair_found;

    pair_found=FALSE;
    while(ptr!=NULL && ptr->next_ptr!=NULL && pair_found==FALSE)
        if(ptr->value==ptr->next_ptr->value)
            pair_found=TRUE;
        else
            ptr=ptr->next_ptr;
    return pair_found;
}
```

□

Es. 2.7.9 Si definisca la funzione c che riceve una lista collegata con puntatori e ne rimuove le ripetizioni di valori consecutivi (e.g. ricevendo in ingresso la lista 1,2,4,4,2,5,4,5,5 deve essere restituita la lista 1,2,4,2,5,4,5). Si osservi che l'implementazione non ha bisogno di un doppio puntatore alla testa della lista perché per definizione del problema il primo elemento della lista non deve essere modificato.

Soluzione: Analogamente a quanto osservato nell'esercizio 2.7.8, la soluzione ha la forma di un visita nella quale la condizione di legalità richiede che siano non nulli sia il puntatore all'elemento corrente che quello al suo successore:

```
void remove_subsequent_pairs(struct list * ptr)
{
    struct list * tmp_ptr;

    while(ptr!=NULL && ptr->next_ptr!=NULL)
    {   if(ptr->value==ptr->next_ptr->value)
        {   tmp_ptr=ptr->next_ptr;
            ptr->next_ptr=ptr->next_ptr->next_ptr;
            free(tmp_ptr);
        }
        else
        {   ptr=ptr->next_ptr;
        }
    }
}
```

□

Es. 2.7.10 Si definisca la funzione *c* che riceve una lista collegata con puntatori e ne rimuove le ripetizioni (e.g. ricevendo in ingresso la lista 1,2,4,4,2,5,4,5,5 deve essere restituita la lista 1,2,4,5).

Soluzione: Se è possibile ordinare la lista, il problema può essere ricondotto al caso dell'esercizio 2.7.9 e risolto invocando prima un algoritmo di ordinamento e poi la funzione `remove_subsequent_pairs()`. Se invece l'ordine della lista non può essere modificato, è possibile risolvere il problema con una visita che per ogni valore incontrato invoca una funzione che rimuove dal resto della lista tutti gli elementi con valore uguale a quello corrente.

```
void remove_pairs(struct list * ptr)
{
    while(ptr!=NULL)
    {
        if(ptr->next_ptr!=NULL)
            remove_value(&(ptr->next_ptr),ptr->value);
        ptr=ptr->next_ptr;
    }
}

void remove_value(struct list ** ptrptr, float value)
/* rimuove dalla lista la cui testa e' puntata da *ptrptr tutti gli
   elementi che contengono un valore uguale a value.
   L'implementazione omette di tenere conto che il test di uguaglianza sui
   float deve trattare il problema della precisione finita. */
{
    while(*ptrptr!=NULL)
    {
        if((*ptrptr)->value==value)
        {
            tmp_ptr=*ptrptr;
            *ptrptr=(*ptrptr)->next_ptr;
            free(tmp_ptr);
        }
        ptrptr=&((*ptrptr)->next_ptr);
    }
}
```

□

Es. 2.7.11 Si valuti la complessità dell'algoritmo proposto come soluzione dell'esercizio 2.7.10.

Soluzione: Se la lista viene ordinata, la complessità delle operazioni di cancellazione è $O(N)$ visto che ogni elemento viene visitato una sola volta e ogni cancellazione ha costo costante. L'operazione di ordinamento può essere eseguita in principio in tempo $O(N \ln(N))$ ma richiede che gli elementi della lista siano

copiatì su un vettore di appoggio per abilitare l'accesso in tempo costante che è necessario per una implementazione efficiente dell'ordinamento.

Se invece la lista deve essere mantenuta nella sua iniziale rappresentazione, nel caso pessimo in cui non ci sono duplicazioni, per ogni elemento della lista è effettuata una visita di tutti i successori, che risulta in un costo totale della forma $\sum_{n=0}^{N-1} n$ e quindi in una complessità $\Theta(N^2)$. \square

Es. 2.7.12 Sono assegnate due sequenze ordinate di numeri float, rappresentate su due liste in forma collegata con puntatori. Si scriva una funzione c che fonde le due liste sulla prima in maniera ordinata.

Soluzione: L'operazione si riduce a una visita sulla seconda lista: ciascun elemento visitato viene inserito sulla lista destinazione.

```
void merge (struct list ** A_ptrptr, struct list * B_ptr )
/* A e B liste ordinate. Inserisce in ordine in A gli elementi di B. */
{
    while(B_ptr!=NULL)
    {   ord_insert(A_ptrptr, B_ptr->value); // inserimento in ordine
        B_ptr=B_ptr->next_ptr;
    }
}

void ord_insert(struct list ** ptrptr, float value);
/* inserimento in testa */
```

\square

Es. 2.7.13 Si valuti la complessità della soluzione proposta per l'esercizio 2.7.12 e si proponga una versione ottimizzata della soluzione.

Soluzione: Siano N_A e N_B le lunghezze delle due liste. Nel caso pessimo in cui tutti gli elementi di B sono maggiori di tutti quelli di A , l'inserimento del $(k+1)-mo$ elemento di B costa $N_A + k$. Sommando si ottiene una forma di tipo $N_B * (N_A + N_B)$.

È possibile ridurre la complessità al tempo $N_A + N_B$ evitando di scandire A a partire dalla testa in tutti gli inserimenti. Per questo occorre che il puntatore Aptrptr sia fatto avanzare parallelamente alla visita di B :

```
void merge_opt (struct list ** A_ptrptr, struct list * B_ptr )
/* A e B liste ordinate. Inserisce in ordine in A gli elementi di B.
   Opera in tempo lineare. */
{
    while(B_ptr!=NULL)
```

```

    {
        while(*Aptrptr!=NULL && (*Aptrptr)->value<B_ptr->value)
            Aptrptr=&((*Aptrptr)->next_ptr);
        pre_insert(Aptrptr,B_ptr->value);
        B_ptr=B_ptr->next_ptr;
    }
}

void pre_insert(struct list ** ptrptr,float value);
/* inserimento in testa */

```

□

→ **Es. 2.7.14** Si definisca la funzione *c* che riceve in ingresso un vettore *V* di *N* valori interi e costruisce una lista istogramma che rappresenta una unica volta ciascun valore del vettore assieme al numero delle sue occorrenze nel vettore stessi. Nel corso della costruzione la lista sia tenuta in ordine rispetto al valore rappresentato.

Soluzione: La funzione *build_histogram* scandisce il vettore e invoca ripetutamente la funzione *add_value* che ha la responsabilità di scandire la lista ricercando il valore: se questo esiste già ne viene incrementato il contatore; altrimenti viene inserito in ordine in lista un nuovo elemento con contatore inizializzato a 1.

```

struct list_h{
    int value;
    int occurrences;
    struct list * next_ptr;
};

void build_histogram_list(int * V, int N, struct list_h ** ptrptr)
{
    int count;

    for(count=0;count<N;count++)
        add_value(ptrptr,V[count]);
}

void add_value(struct list_h ** ptrptr, int value)
{
    while(*ptrptr!=NULL && (*ptrptr)->value>value)
        ptrptr=&((*ptrptr)->next_ptr);

    if(*ptrptr!=NULL && *ptrptr->value==value)
        (*ptrptr)->occurrences++;
    else
        pre_insert_h(ptrptr,value);
}

```

```

void pre_insert_h(struct list ** ptrptr, int value);
{ struct list_h * tmp_ptr;

    tmp_ptr=*ptrptr;
    *ptrptr=(struct list *)malloc(sizeof(struct list_h));
    (*ptrptr)->value=value;
    (*ptrptr)->occurrences=1;
    (*ptrptr)->next_ptr=tmp_ptr;
}

```

□

Es. 2.7.15 Si calcoli la complessità della soluzione proposta per l'esercizio 2.7.14 e se ne discuta la possibile ottimizzazione.

Soluzione: Nel caso pessimo di un vettore con elementi tutti distinti e ordinati, per ogni elemento del vettore è necessario scandire l'intera lista ed effettuare un inserimento in coda. Questo richiede N inserimenti e un numero di confronti proporzionale a $\sum_{n=1}^{N-1} n = \frac{N*(N-1)}{2}$, che risulta in una complessità $O(N^2)$.

Una soluzione più efficiente si ottiene ordinando il vettore in tempo $O(N \ln(N))$ e poi costruendo la lista in tempo lineare. Per costruire la lista in tempo lineare occorre effettuare gli inserimenti in testa, il che si può ottenere ad esempio scandendo il vettore in ordine inverso. □

Es. 2.7.16 Si definisca la funzione c che riceve in ingresso due sequenze di interi $s1$ e $s2$ rappresentate come liste in forma collegata con puntatori e verifica se $s1$ è una sottosequenza di $s2$.

Soluzione: Lo schema della soluzione ricalca quella dell'esercizio 1.6.7, qui applicato a una lista piuttosto che a un vettore. Per gestire la maggiore complessità della scansione conviene scomporre l'implementazione in due funzioni.

```

Boolean scan4match(struct list * ptr1, struct list * ptr2)
/* verifica se la lista ptr1 contiene a qualche offset la lista ptr2 */
{
    Boolean found;

    found=FALSE;
    while(ptr1!=NULL && found==FALSE)
    {   if(match(ptr1,ptr2)==TRUE)
        found=TRUE;
        else
            ptr1=ptr1->next_ptr;
    }
}

```

```

    return(found);
}

Boolean match(struct list * ptr1, struct list * ptr2)
/* restituisce TRUE se il prefisso di ptr1 e' uguale al prefisso di ptr2 */
{
    Boolean match;

    match=TRUE;
    while(ptr2!=NULL && match==TRUE)
    {   if(ptr1==NULL || ptr2->value!=ptr1->value)
        match=FALSE;
        else
        {   ptr2=ptr2->next_ptr;
            ptr1=ptr1->next_ptr;
        }
    }
    return(match);
}

```

□

Es. 2.7.17 Si consideri il caso in cui insiemi di valori interi siano rappresentati come liste in forma collegata con puntatori, e si assuma che le liste siano mantenute in ordine.

Si definiscano due funzioni c che ricevono in ingresso due insiemi e costruiscono rispettivamente la rappresentazione dell'insieme intersezione, unione, e unione esclusiva (xOr).

Soluzione:

```

void intersezione(struct list* ptr1, struct list* ptr2, struct list** ptrptr)
{
    while(ptr1!=NULL)
    {   if(ord_search(ptr2, ptr1->value)==TRUE)
        ord_insert(ptrptr,ptr1->value);
        ptr1=ptr1->next_ptr;
    }
}

void unione(struct list * ptr1, struct list * ptr2, struct list ** ptrptr)
{
    copy(ptr1,ptrptr);
    while(ptr2!=NULL)
    {   if(ord_search(ptr1, ptr2->value)==FALSE)
        ord_insert(ptrptr,ptr2->value);
        ptr2=ptr2->next_ptr;
    }
}

```

```

}

void xOr(struct list * ptr1, struct list * ptr2, struct list ** ptrptr)
{
    init(ptrptr);
    while(ptr1!=NULL)
    {   if(ord_search(ptr2, ptr1->value)==FALSE)
        ord_insert(ptrptr,ptr1->value);
        ptr1=ptr1->next_ptr;
    }
    while(ptr2!=NULL)
    {   if(ord_search(ptr1, ptr2->value)==FALSE)
        ord_insert(ptrptr,ptr2->value);
        ptr2=ptr2->next_ptr;
    }
}

Boolean ord_search(struct list * ptr, int value)
/* ricerca value sulla lista puntata da ptr.
   Restituisce TRUE in caso di successo, FALSE altrimenti.
   Assume che la lista sia ordinata. */
{
    Boolean found;

    found=FALSE;
    while(found==FALSE && ptr!=NULL && ptr->value<=value)
    {
        if(ptr->value==value)
            found==TRUE;
        else
            ptr=ptr->next_ptr;
    }
    return found;
}

void insert(struct list ** ptrptr, int value);
/* inserisce value nella lista puntata da *ptrptr.
   Assume che la lista sia in ordine e opera l'inserimento in ordine */

void copy(struct list **ptrptr, struct list * ptr);
/* copia ptr su ptrptr mantenendo l'ordine */

void init(struct list **ptrptr);
/* inizializza la lista puntata da *ptrptr */

```



Es. 2.7.18 Si valuti la complessità della soluzione proposta per l'esercizio 2.7.17 e si delinei un approccio per la sua ottimizzazione.

Soluzione: Siano N_1 e N_2 le lunghezze delle due liste in ingresso. Nella funzione che costruisce l'intersezione, il costo sulle chiamate a `search()` è $O(N_1 * N_2)$. Il costo sulle chiamate a `insert()` è $O(\frac{N_1*(N_1+N_2)}{2})$. La complessità totale è quindi $O(N_1*(N_1+N_2))$. Nel caso della unione occorre aggiungere il costo iniziale della copia effettuata da `copy()` che può essere realizzata in tempo $O(N_2)$ e non modifica quindi la complessità della soluzione.

È possibile ridurre la complessità all'ordine $O(N_1 + N_2)$ scorrendo parallelamente le due liste sorgente e la lista destinazione e usando il fatto che entrambe sono in ordine. \square

Es. 2.7.19 Si definisca la funzione `c` che riceve in ingresso un array W di M records, ciascuno composto di un intero `fkey` e di un dato strutturato `y` (il cui formato non è rilevante ai fini dell'esercizio), e costruisce la lista (in formato collegato con puntatori) degli indici del records il cui campo `fkey` è uguale a un valore `key` assegnato.

Soluzione:

```

struct record{
    struct data y;
    int fkey;
};

void relation2object(struct record *W, int M, struct list **ptrptr, int key)
{
    int count;

    init(ptrptr);
    for(count=0;count<M;count++)
        if(W[count].fkey==key)
            pre_insert(ptrptr,count);
}

void pre_insert(struct list **ptrptr, int value);
/* inserimento in testa */

void init(struct list **ptrptr);
/* inizializzazione */

```

\square

Es. 2.7.20 Si consideri il caso del sistema informativo per la gestione di un parcheggio al quale accedono autoveicoli autorizzati da un insieme di dipartimenti. Ciascun dipartimento è rappresentato da un descrittore che include il numero di

telefono della segreteria del dipartimento e una lista di targhe automobilistiche autorizzate.

Si definiscano le strutture che rappresentano il descrittore del dipartimento e la lista delle sue targhe. Si definisca la funzione c che riceve in ingresso un vettore di descrittori di dipartimento e un numero di targa e verifica che essa sia registrata almeno su un dipartimento, restituendo in caso positivo il numero di telefono del dipartimento a cui è associata la targa.

Soluzione: L'informazione è codificata su un vettore V[] di records di tipo struct Dept, ciascuno dei quali include un numero di telefono e il puntatore a una lista di numeri di targhe.

La funzione search_dept() effettua una scansione di ricerca sul vettore dei dipartimenti V[]: è un ciclo while di cui il booleano found e il contatore count codificano le condizioni di arresto per terminazione e legalità. Per ogni descrittore di dipartimento V[count], la funzione search_shield() effettua a sua volta una ricerca (del tutto convenzionale) sulla lista delle targhe del dipartimento stesso; si osservi che tale lista viene referenziata come V[count].ptr). Se la targa cercata viene trovata, il telefono richiesto dal problema viene selezionato sulla struttura del dipartimento corrente come V[count].phone_number.

```
struct Dept{
    int phone_number;
    struct list * ptr;
    ...
};

struct list{
    int number
    struct list * next_ptr;
};

Boolean search_dept(struct Dept* V,int N,int shield_number,int* phone_ptr)
{   Boolean found;

    count=0;
    found=FALSE;
    while(found==FALSE && count<N)
    {   if(search_shield(V[count].ptr,shield_number))
        {   *phone_ptr=V[count].phone_number;
            found=TRUE;
        }
        else
        {   count++;
        }
    }
    return found;
}
```

```

Boolean search_shield(struct list * ptr, int shield_number)
{   Boolean found;

    found=FALSE;
    while(found==FALSE && ptr!=NULL)
    {   if(ptr->number==shield_number)
        found=TRUE;
        else
            ptr=ptr->next_ptr;
    }
    return found;
}

```

□

Es. 2.7.21 Si consideri il caso di una anagrafe di cittadini rappresentati su un vettore di records. Ciascun record è composto da un nome un cognome e una lista di parentele. A sua volta ciascuna parentela è rappresentata con un intero che codifica il tipo di parentela e con un indice che codifica la posizione nel vettore di anagrafe del cittadino parente.

Si definiscano le strutture e le funzioni impiegate nella rappresentazione e la funzione che riceve in ingresso il nome e cognome di un cittadino e stampa il nome e cognome dei suoi parenti e la relazione di parentela che li lega.

Soluzione: L'informazione è codificata su un vettore R[] di records di tipo struct residente, ciascuno dei quali include nome, cognome e codice di un residente e la testa ptr di una lista di relazioni di parentela. A sua volta ciascun elemento di tale lista include un descrittore del tipo di parentela e un indice che identifica la posizione nel vettore R[] del descrittore del parente.

La funzione print_parenti() effettua una ricerca sul vettore dei residenti R[] comparando il nome e il cognome cercato contro quelli di ciascun descrittore R[count] visitato; si osservi che la comparazione è basata sulla funzione strcmp() inclusa nella libreria standard string.h. Nel caso in cui venga ripetuto un descrittore V[count] con nome e cognome corrispondenti al target della ricerca, i descrittori dei parenti sono stampati con una visita sulla lista delle parentele V[count].ptr. Tale visita è effettuata dalla funzione print_parenti() che riceve anche il vettore dei descrittori R. La visita ha una struttura convenzionale, ma è utile soffermarsi sul riferimento alle variabili nell'argomento dell'operazione printf(): ptr è il puntatore all'elemento corrente nella lista delle parentele e ptr->type è il tipo di parentela; ptr->indice_parente è l'indice in R[] del descrittore del parente la cui parentela è puntata da ptr; R[ptr->indice_parente].nome è quindi il campo nome sul descrittore del parente.

```

struct residente{
    char nome[81];
    char cognome[81];
    int codice;
    struct parentela * ptr;
};

struct parentela{
    int tipo;
    int indice_parente;
    struct parentela * next_ptr;
};

Boolean print_parenti(struct residente * R, int N, char * nome, char * cognome)
{
    found=FALSE;
    count=0;
    while(count<N && found==FALSE)
    {   if(strcmp(R[count].nome,nome)==0 && strcmp(R[count].cognome,cognome)==0)
        found=TRUE;
        else count++;
    }
    if(found==TRUE)
        print_parenti(R[count].ptr,R);
    else
        printf("\n non esiste il residente %s %s", nome, cognome);

    return found;
}

void print_parenti(struct parentela * ptr, struct residente * R)
/* visita di stampa sui parenti.
   Per ciascuno stampa il tipo di parentela e il nome del parente */
{
    while(ptr!=NULL)
    {   printf("\n %s %s e' parente di tipo %d",
            R[ptr->indice_parente].nome,R[ptr->indice_parente].cognome,ptr->type);
        ptr=ptr->next_ptr;
    }
}

```

□

Es. 2.7.22 Si consideri la seguente rappresentazione di una rete ferroviaria: i nodi della rete sono rappresentati su un vettore di records, ciascuno composto da un nome e una lista di collegamenti; ciascun collegamento rappresentato da una lunghezza e da un indice intero che codifica la posizione nel vettore della stazione collegata.

Si definiscano le strutture e impiegate nella rappresentazione e la funzione che riceve in ingresso il nome di una stazione e stampa a video il nome e la distanza delle stazioni ad essa collegata.

Soluzione: La soluzione ha circa la stessa struttura di quella dell'esercizio 2.7.21.

```
struct stazione{
    char nome[81];
    struct collegamento * link_ptr;
};

struct collegamento{
    int lunghezza;
    int indice_arrivo;
    struct collegamento * next_ptr;
};

Boolean print_stazioni_collegate(struct stazione * S, int N, char * nome)
/* ricerca sul vettore S una stazione con il nome assegnato;
   Se non lo trova restituisce FALSE, altrimenti restituisce TRUE
   e stampa i nomi delle stazioni collegate */
{
    boolean found;
    int count;

    found=FALSE;
    count=0;
    while(count<N && found==FALSE)
    {   if(strcmp(S[count].nome,nome)==0)
        found=TRUE;
        else
            count++;
    }

    if(found==TRUE)
        print_stazioni_raggiunte(S[count].link_ptr);
    else
        printf("\n non esiste una stazione con nome %s", nome);

    return found;
}

void print_stazioni_raggiunte(struct collegamento *link_ptr, struct stazione *S)
/* visita la lista di collegamenti, per ciascuno stampa la lunghezza e
   il nome della stazione raggiunta */
{   while(link_ptr!=NULL)
    {   printf("\n %s a distanza %d",
            S[link_ptr->indice_arrivo].nome, link_ptr->lunghezza);
        link_ptr=link_ptr->next_ptr;
    }
}
```

}

□

Es. 2.7.23 È assegnato un array V di N records che rappresentano informazione di natura anagrafica. Ciascun record è composto di un nome, un cognome, un intero key che identifica in modo univoco il soggetto descritto, un intero parentkey che contiene il codice identificativo del padre del soggetto descritto. Si vuole costruire un array W che rappresenta la stessa informazione attraverso un array di N records ciascuno dei quali descrive un soggetto riportandone il nome, il cognome, e la lista degli indici dei records che descrivono i figli.

Si definiscano le strutture dati del c che rappresentano i records di V e W e la funzione c che riceve in ingresso il vettore V e provvede ad allocare, costruire e restituire il vettore W .

Soluzione:

```
struct Vrecord
{ int key;
  char nome[81];
  char cognome[81];
  int parentkey;
};

struct Wrecord
{ char nome[81];
  char cognome[81];
  struct list * children_ptr;
};

struct list
{ int child_index;
  struct list * next_ptr;
}

void relation2object(struct Vrecord * V, int N, struct Wrecord ** Wptr)
{ int count;

  *Wptr=(struct Wrecord *)malloc(N*(sizeof(struct Wrecord)));

  for(count=0;count<N;count++)
  {
    strcpy(V[count].nome,W[count].nome);           // copia il primo argomento
                                                    // sul secondo
    strcpy(V[count].cognome,W[count].cognome); // idem

    init(&W[count].children_ptr);
  }
}
```

```

        for(count2=0;count2<N;count2++)
    {
        if(V[count2].parentkey==V[count].key)
            pre_insert(&W[count].children_ptr,count2);
    }
}

void pre_insert(struct list ** ptrptr, int child_index);
/* inserimento in testa*/

```

□

Es. 2.7.24 Si valuti la complessità della soluzione proposta per il problema dell'esercizio 2.7.23.

Soluzione: per ogni elemento di W vengono eseguite un numero costante di operazioni e viene scandito completamente V. Nella scansione ciascun elemento di V viene eseguito un numero costante di operazioni. Dunque la complessità è $O(N^2)$ □

Es. 2.7.25 Si consideri una struttura di informazione composta di docenti, insegnamenti e registri: ciascun docente è descritto da un nome, un cognome, e una lista di insegnamenti; ciascun insegnamento ha un codice e un riferimento a un registro; un registro contiene un numero di lezioni previste e un numero di lezioni svolte.

Si definiscano: le strutture c che rappresentano l'informazione; la funzione che riceve in ingresso un vettore di docenti e un codice di insegnamento e restituisce il numero di lezioni memorizzate sul registro di quell'insegnamento; il frammento del codice del chiamante che invoca la funzione e dichiara le variabili coinvolte, e la organizzazione complessiva del programma, completo di prototipi e eventuali direttive Per semplicità si assume che non esistano due lezioni con lo stesso codice tenute da docenti diversi.

Soluzione:

```

struct docente{
    char nome[81];
    char cognome[81];
    struct lista_ins * ptr;
};

struct lista_ins{
    int codice;
    struct registro * reg_ptr;

```

```

    struct lista_ins * next_ptr;
};

struct registro {
    int lezioni_previste;
    int lezioni_svolte;
};

#define N 128
void main(void)
{   struct docente V[N];
    int codice;
    int lezioni;
    // ...
    lezioni=conta_lezioni(V, N, codice);
    // ...
}

int conta_lezioni(struct docente * V, int n, int codice)
{
    int count;
    int lezioni;
    Boolean found;

    lezioni=0;
    count=0;
    found=FALSE;

    while (count<n && found==FALSE)
    {   lezioni=conta_suUnDocente(V[count].ins_ptr,codice);
        if(lezioni!=0)
            found=TRUE;
        else
            count++;
    }
    return lezioni;
}

int conta_suUnDocente(struct lista_ins * ptr, int codice)
{
    int lezioni;
    Boolean found;

    lezioni=0;
    found=FALSE;
    while(ptr!=NULL && found==FALSE)
    {   if(ptr->codice==codice)
        {   found=TRUE;
            lezioni=ptr->reg_ptr->lezioni_svolte;
        }
    }
}

```

```

    }else
        ptr=ptr->next_ptr;
}
}

```

□

Es. 2.7.26 Si consideri il problema di determinare l'ultimo treno utile con il quale è possibile partire da una locazione *START* per raggiungere la locazione *TARGET* entro un'ora assegnata usando una coincidenza presso la locazione *COMMUTE* (e.g. trovare l'ultimo treno con il quale si parte da Firenze per arrivare ad Ancona entro le 11 cambiando a Bologna). In ingresso sono assegnate due sequenze che codificano l'orario delle corse da *START* a *COMMUTE* e da *COMMUTE* a *TARGET*. Ciascuna corsa è caratterizzato da un numero identificativo, un'ora di partenza, un'ora di arrivo, e un tipo.

Si definisca la struttura dati che rappresenta una sequenza di corse. Si definisca la funzione *c* che riceve in ingresso due sequenze di corse *START*→*COMMUTE* e *COMMUTE*→*TARGET* e un ora di arrivo *DEADLINE* e fornisce in uscita l'ultima possibile ora di partenza da *START* per raggiungere *TARGET* entro *DEADLINE*.

Per semplicità si assuma di potere rappresentare l'ora come numero intero. Si assume anche che non siano possibili sorpassi su nessuna delle tratte e che non siano accettabili soluzioni in cui non si arriva a destinazione entro la giornata.

Soluzione:

```

struct list {
    int start_time;
    int dest_time;
    int type;
    struct list * next_ptr;
};

int chain(struct list * first_hop_ptr, struct list * second_hop_ptr, int deadline)
/* restituisce l'ultimo possibile tempo di partenza.
   Se non ci sono soluzioni restituisce -1. */
{
    int last_commute_time;

    last_commute_time=find_last_start_time(second_hop_ptr, deadline);
    return find_last_start_time(first_hop_ptr, last_commute_time)
}

int find_last_start_time(struct list * ptr, int deadline)
/* restituisce 'ora dell'ultimo treno in lista che arriva a destinazione
   entro deadline.
   Se non ci sono soluzioni restituisce -1.

```

```

Assume che la lista sia ordinata rispetto allo start_time
(e quindi in assenza di sorpassi anche rispetto al dest_time) */
{ int last_start_time;

    last_start_time=-1;
    while(ptr!=NULL&& ptr->dest_time<deadline)
    {   last_start_time =ptr->start_time;
        ptr=ptr->next_ptr;
    }
    return last_start_time;
}

```

□

Es. 2.7.27 Si consideri il caso di un orario ferroviario rappresentato su un vettore di records, ciascuno composto di un ora di partenza e di una sequenza di fermate, ciascuna composta dal nome della stazione e l'ora di arrivo, rappresentate su una lista collegata con puntatori.

Per semplicità si assuma che le ore di partenza e arrivo possano essere rappresentate come numeri interi, che il numero 25 sia maggiore di qualsiasi ora esistente, e che il nome di una stazione sia rappresentato come array di 81 caratteri.

Si definiscano le strutture dati che rappresentano l'informazione relativa ad un treno e alla sua lista di fermate. Si definisca la funzione che restituisce l'ultima ora a cui è possibile partire per arrivare ad una assegnata stazione entro un assegnato tempo di arrivo. Si definisca la funzione main() che include la dichiarazione statica del vettore orario e invoca la funzione sopra menzionata.

Soluzione:

```

#define N 128
#define MAXIMUM_TIME 25

struct list{
    char station[81];
    int arrive_time;
    struct list * next_ptr;
};

struct train{
    int start_time;
    struct list * ptr;
};

void main(void)
{ struct record Orario[N];
    char destination [81];
    int last_start_time;

```

```

int arrive_time;

scanf("%s",destination);
scanf("%d",arrive_time);
last_start_time=get_last_start_time (Orario,N,destination,arrive_time);
if(last_start_time!=MAXIMUM_TIME)
    printf("last start time: %d", last_start_time);
else
    printf("no connection available! ");
}

int last_start_time(struct record * Orario, int size, char * destination,
                     int arrive_time)
/* restituisce l'ultima ora di partenza di un treno che arriva a destination
   entro arrive_time. Se non esiste alcun treno che arriva (in tempo) a destination
   restituisce MAXIMUM_TIME. */
{ int count;
  int _last_start_time

  for(count=0,_last_start_time=MAXIMUM_TIME; count<size; count++)
  { start_time =get_to_destination_in_time
    (Orario[count],destination,arrive_time,arrive_time);
    if(start_time > _last_start_time)
      _last_start_time=start_time;
  }
  return _last_start_time;
}

int get_to_destination_in_time(struct record treno,char * destination,
                               int arrive_time)
/* restituisce MAXIMUM_TIME se destination non e' nella lista delle fermate.
   Altrimenti restituisce l'ora di partenza del treno */
{ struct list * ptr;

  found=FALSE;
  ptr=treno.ptr;
  while(found==FALSE)
  { if(strcmp(ptr->destination,destination)==0)
    found=TRUE;
    else
      ptr=ptr->next_ptr;
  }
  if(found==TRUE && ptr->arrive_time<arrive_time)
    return treno.start_time;
  else
    return MAXIMUM_TIME;
}

```



... ripetuta, ripetutamente !

Capitolo 3

Estensione dal c al c++

In questa Appendice viene sinteticamente descritto l'incremento dal c al c++:

- Legame dei parametri per riferimento e riferimenti costanti
- Classi e oggetti
- Attributi e metodi statici
- Ereditarietà
- Polimorfismo e dynamic binding

3.1 Legame per riferimento e riferimenti costanti

Mentre in c parametri attuali e formali sono sempre legati per valore, il c++ permette anche l'uso del legame per riferimento. In questo caso, i riferimenti a un parametro formale all'interno di una funzione colpiscono il parametro attuale definito dal chiamante. Si osservi che questo implica che dal punto di vista sintattico un parametro attuale legato per riferimento non ha più la forma generale di un'espressione ma piuttosto quella di un riferimento.

Una stessa funzione può combinare parametri legati per valore e per riferimento. Questi ultimi sono caratterizzati nella lista dei parametri formali per il fatto di avere il nome preceduto dal simbolo di ampersand &. Nell'esempio che segue, i parametri a e b sono legati per valore, mentre c è legato per riferimento. Questo permette alla funzione `sum()` di modificare la variabile z nello spazio degli indirizzi del chiamante:

```
void client(void)
{   int x;
    int y;
```

```

    int z;
    sum(x,y,z);
}

Void sum(int a, int b, int &c)
{   c = a+b;
}

```

Un riferimento può essere dichiarato *costante* attraverso il qualificatore `const`, per impedire alla funzione chiamata di modificarne il valore. Questo permette di coniugare l'efficienza nel carico dello stack con l'efficacia nel disaccoppiamento tra le funzioni chiamante e chiamata, realizzando un meccanismo di *minimo privilegio* concesso alla funzione chiamata.

Nell'esempio che segue, la funzione `put()` riceve il parametro `v` per riferimento. Alla chiamata, la variabile `value` viene legata come parametro attuale a `v` ma il suo uso è limitato ad essere costante; se nella sua definizione `put()` includesse espressioni che producono side-effects su `value`, il compilatore riporterebbe un errore.

```

void put(const float &v);

void client(void)
{   float value;
    put(value);
}

```

3.2 Classi e oggetti

La *classe* del C++ estende il concetto di variabile strutturata del C. Come una struttura, una classe raccoglie un insieme di variabili (denominati attributi o anche *data members*) che divengono referenziabili attraverso un nome collettivo e un identificativo simbolico. In aggiunta, la classe può anche includere un insieme di funzioni (denominati metodi o anche *function members*) che operano nel contesto della classe stessa. Come già avveniva per le strutture, una classe definisce un tipo, del quale possono essere create una varietà di istanze (denominate *oggetti*).

Nella costruzione e nell'uso di una classe intervengono la *definizione* della classe e dei metodi che vi compaiono, la *dichiarazione statica* e/o la *allocazione dinamica* di *oggetti* che istanziano la classe, e il *riferimento* a membri di oggetti istanziati. La definizione dei metodi è più spesso denominata *implementazione*.

3.2.1 Definizione

La definizione di una classe combina i concetti già introdotti separatamente nella definizione di strutture e nella dichiarazione di funzioni del C.

Per fare leva su concetti già acquisiti, consideriamo l'implementazione di una struttura stack. In c, questa viene realizzata attraverso la definizione di una struct stack e la dichiarazione e definizione di un insieme di funzioni che ne manipolano le istanze, tipicamente init(), push() e pop():

```
struct stack
{ int TOS;
  data * buffer;
  int size;
};

boolean push(struct stack * ptr, data value)
{ ...
}

boolean pop(struct stack * ptr, data * value_ptr)
{ ...
}

void init(struct stack * ptr, int size)
{ ...
}
```

La definizione della struttura stack e la definizione delle funzioni init(), push() e pop() sono tra loro fortemente accoppiate in quanto una modifica su ciascuno dei quattro elementi ha probabilmente effetti sugli altri. Tuttavia, di questo accoppiamento il programmatore può rendere evidenza solo attraverso la disciplina di tenere le dichiarazioni delle funzioni prossime alla definizione della struttura.

Con l'introduzione della classe, realizzata attraverso il costrutto class, le variabili che formano lo stack sono invece dichiarate entro un unico contenitore assieme alle funzioni che le manipolano. Questo realizza bene il concetto di tipo composto congiuntamente da un insieme di valori (lo spazio dei possibili valori degli attributi) e da un insieme di operazioni che ci si applicano sopra (l'insieme delle operazioni implementate dai metodi):

```
class stack
{ int TOS;
  data * buffer;
  int size;

  boolean push(data value);
  boolean pop(data * value_ptr);
  void init(int size);
};
```

I metodi dichiarati in una classe sono definiti (più spesso si dice *implementati*) separatamente, mantenendo la sintassi delle funzioni del c, salvo che in questo caso il nome della funzione è preceduto dal nome della classe di appartenenza. Ad esempio, la funzione push() che era stata dichiarata nella classe stack è definita come:

```

boolean stack::push(data value)
{
    if(TOS==size)
        return FALSE;
    else
    {   buffer[TOS]=value;
        TOS++;
        return TRUE;
    }
}

```

Gli attributi di una classe sono visibili nel corpo dei metodi della classe stessa (ovvero un metodo può esprimere un riferimento ad un membro della classe senza che questo figuri tra i suoi parametri formali). Nell'esempio questo è il caso degli attributi `buffer`, `size` e `TOS` che sono referenziati nell'implementazione di `push()` pur non figurando tra i parametri formali del metodo.

Questa regola di visibilità conferisce agli attributi di una classe il significato di un contesto condiviso tra i metodi. Rispetto al c, questo costituisce un fattore di maggiore semplificazione degli schemi di programmazione, riducendo fortemente la complessità delle interfacce e la necessità di validazione dei parametri su cui operano i metodi stessi. In negativo, la condivisione degli attributi dell'oggetto aumenta l'*accoppiamento* tra i metodi della classe, richiedendo che il loro sviluppo e manutenzione siano affrontati in maniera unitaria. Si tratta di un problema di minore impatto, ampiamente compensato dai vantaggi, purché la classe sia progettata in modo da raccogliere un insieme di *responsabilità cohesive*.

3.2.2 Visibilità public, protected, private

Mentre i membri di uno `struct` sono tutti indistintamente visibili non appena sia disponibile un riferimento alla struttura, nella definizione di una classe è possibile differenziare la visibilità dei diversi membri attribuendo a ciascuno di essi una di tre modalità: `public`, `private`, `protected`:

- I membri dichiarati nella clausola `public` sono visibili a chiunque disponga dell'indirizzo o del riferimento all'oggetto;
- I membri nella clausola `private` sono visibili solo nei metodi della classe stessa;
- I membri `protected` sono visibili solo nei metodi della classe stessa o di una classe da essa derivata secondo il meccanismo di ereditarietà descritto più avanti.

Nell'esempio che segue, gli attributi `TOS`, `buffer` e `size` sono referenziabili solo nei metodi della classe `stack` o di sue derivazioni per ereditarietà, mentre i metodi `push()`, `pop()`, e `init()` sono referenziabili nel corpo di qualsiasi funzione disponga dell'indirizzo o del riferimento all'oggetto di tipo `stack`:

```
class stack
{   protected:
    int TOS;
    data * buffer;
    int size;
public:
    void init(int s);
    boolean push(data value);
    boolean pop(data * value_ptr);
};
```

La possibilità di differenziare la visibilità dei diversi membri di una classe gioca un ruolo primario nel *disaccoppiamento* tra le classi, costituendo un meccanismo di *incapsulamento* fondamentale per la manutenibilità e l'affidabilità di una applicazione.

3.2.3 Overloading

Nella definizione di una classe è possibile definire più funzioni con lo stesso nome e lo stesso tipo restituito, ma con una sequenza di parametri formali (comunemente detta *signature*) diversa nel numero e/o nel tipo dei parametri. Si dice in questo caso che il nome della funzione è *overloaded*, e questo costituisce una realizzazione del concetto per cui una stessa operazione astratta può avere più implementazioni concrete capaci di lavorare su dati diversi. In ultimo, questo è lo stesso concetto per cui si denota con lo stesso simbolo + l'operazione di somma nelle diverse implementazioni riservate al trattamento di valori `int` e `float`.

Il compilatore associa a ciascuna diversa definizione un diverso segmento di codice e un diverso simbolo ottenuto estendendo il nome della funzione con un suffisso che identifica in modo univoco i tipi nella signature. Al momento della compilazione di un riferimento alla funzione, l'indirizzo della funzione che deve essere legata viene determinato in base alla signature dei parametri attuali.

Si intuisce che il meccanismo di legame basato sui tipi dei parametri attuali attribuisce al controllo dei tipi stessi un valore molto più forte di quanto avvenga invece nel c: in c++ un cast sul tipo di un parametro attuale può determinare quale tra varie implementazioni di una funzione viene collegata alla chiamata. Per questa maggiore rilevanza il compilatore del c++ effettua un *type-checking* molto più severo: incongruenze sui tipi che nel c sono trattate come warnings nel c++ risultano invece in errori.

Ad esempio, nella classe `stack` possiamo aggiungere una seconda implementazione dell'operazione di `init()` che oltre alle operazioni già previste provvede anche ad inizializzare le variabili nel buffer ad un valore passato come secondo parametro. La scelta tra le due implementazioni è a carico della funzione che usa l'oggetto di tipo `stack` ed è determinata dal numero e tipo dei parametri attuali specificati nella chiamata:

```
class stack
{ protected:
    int TOS;
    data * buffer;
    int size;
public:
    boolean push(data value);
    boolean pop(data * value_ptr);
    void init(int size);
    void init(int size, data value);
};
```

3.2.4 Creazione

Una volta definita una classe è possibile crearne istanze, dette *oggetti*, nello stesso modo in cui in c è possibile creare variabili nei tipi di base o in loro aggregazioni strutturate. Come per il c, la creazione può avvenire attraverso un meccanismo di istanza statica o dinamica.

L'istanza statica si ottiene dichiarando all'interno di una funzione un oggetto in una classe. Nell'esempio che segue, un oggetto di classe `stack` viene istanziato staticamente assegnandogli nome `S`:

```
void client(void)
{
    ...
    stack S;
    ...
}
```

Gli oggetti istanziati staticamente sono allocati sullo stack di sistema (da non confondere con lo stack applicativo a cui fa riferimento l'esempio) e hanno il tempo di vita della funzione nella cui località sono dichiarati.

Come per una variabile strutturata del c, gli attributi dell'oggetto sono codificati in locazioni contigue, salvo effetti di ottimizzazione che possono aggiungere bytes non usati o modificare l'ordine degli attributi per mantenere le variabili allineate ai multipli di 2, 4 o 8 bytes. Il codice dei metodi non è rappresentato nell'oggetto, essendo invece rappresentato sul segmento di codice del programma per essere condiviso tra tutte le diverse possibili istanze della classe a cui l'oggetto

appartiene. Per ciascun metodo l'oggetto contiene però l'indirizzo della locazione a partire dalla quale è collocata la funzione che implementa il metodo stesso.

La creazione dinamica riproduce sostanzialmente l'uso della funzione `malloc()`, salvo una semplificazione nella apparenza sintattica basata sull'uso dell'operatore `new`:

```
void client(void)
{   stack * sptr;
...
sptr = new stack;
...
}
```

Come avviene per le variabili del C allocate con `malloc()`, gli oggetti creati con `new` sono allocati sullo heap dei dati e restano in vita fino a che il programma termina oppure fino a che essi sono esplicitamente rilasciati. Questo avviene passando all'operatore `delete` l'indirizzo dell'oggetto da rilasciare, in modo del tutto analogo a quanto avviene per la funzione `free`:

```
void client(void)
{   stack * sptr;
...
sptr = new stack;
...
delete sptr;
...
}
```

3.2.5 Costruttori e distuttore

Nella definizione della classe è possibile dichiarare un metodo costruttore che viene automaticamente invocato alla creazione di ogni nuovo oggetto nella classe, sia nelle istanze statiche che nelle creazioni dinamiche. Tale metodo assume tipicamente la responsabilità di inizializzare gli attributi dell'oggetto ed è contraddistinto dal fatto di avere lo stesso nome della classe e non avere un tipo di ritorno.

Il costruttore può essere overloaded in più implementazioni purchè queste abbiano diversa signature. Nel caso non sia definito alcun costruttore, ne viene generato di default uno che non riceve alcun parametro. Il costruttore senza parametri, che chiameremo costruttore void, sia esso definito esplicitamente o di default, è usualmente invocato senza parentesi tonda (i.e. si scrive `stack S;` anziché `stack S();`). I costruttori sono tipicamente nella clausola `public`. Nella programmazione avanzata può avvenire che essi siano inseriti nella clausola `protected` per impedire che siano create istanze della classe e sia invece ammesso creare istanze

di una classe derivata. L'uso di un costruttore `private` o `protected` in congiunzione con metodi statici permette di realizzare un particolare schema noto come *singleton*.

In modo analogo una classe ha anche un metodo *distruttore* che è denotato con il nome della classe preceduto da una tilde (il carattere '~' corrispondente al 126 nella tabella ASCII) che viene automaticamente invocato alla distruzione dell'oggetto. Il distruttore non può essere overloaded e ammette una unica implementazione che non riceve parametri di ingresso.

In riferimento all'esempio dello stack, i costruttori assumono la responsabilità delle operazioni di inizializzazione. Ne possiamo definire due versioni: il primo riceve la dimensione dello stack e provvede alla allocazione del buffer nella dimensione specificata; il secondo riceve anche un valore con il quale inizializza le variabili nel buffer. Il distruttore provvede a rilasciare la memoria:

```
class stack {
    protected:
        int TOS;
        data * buffer;
        int size;
    public:
        stack(int s);           // costruttore
        stack(int s, data v);  // costruttore
        ~stack(void);          // distruttore
        boolean push(data value);
        boolean pop(data * value_ptr);
};

stack::stack(int s)
/* costruttore: alloca il buffer e inizializza le variabili TOS e size */
{
    TOS=0;
    size=s;
    buffer = new data[s];
}

stack::stack(int s, data v)
/* costruttore: alloca il buffer, inizializza le variabili TOS e size,
   inizializza al valore v le variabili sul buffer */
{
    int count;
    TOS=0;
    size=s;
    buffer = new data[s];
    for(count=0;count<s;count++)
        buffer[count]=v;
```

```

}

stack::~stack(void)
/* distruttore: rilascia la memoria del buffer */
{
    delete buffer;
}

```

Come per qualsiasi altro metodo overloaded, la scelta tra i diversi costruttori è a carico del codice chiamante ed è determinata in base al numero e tipo dei parametri attuali specificati nel riferimento al costruttore:

```

void client(void)
{
    stack S;           // viene invocato il costruttore di default
    stack S2(10);     // viene invocato il costruttore con un
                      // parametro legandogli il valore 10
    stack S3(10,7.);  // viene invocato il costruttore con due parametri
                      // legando loro i parametri 10 e 7.
    stack * Sptr;
    Sptr = new stack(10); // viene invocato il costruttore con un
                          // parametro legandogli il valore 10
    Sptr = new stack(10,7.); // viene invocato il costruttore con due
                           // parametri legando i valori 10 e 7.
    ...
}

```

Nella definizione dei costruttori, l'inizializzazione delle variabili nell'oggetto può essere specificata in una modalità che semplifica il lavoro del compilatore, produce implementazioni di maggiore efficienza, e anche risolve alcuni problemi incontrati nella inizializzazione di oggetti che contengono altri oggetti tra i loro attributi. La forma è quella di una funzione che ha il nome della variabile inizializzata e assume come argomento una espressione che restituisce il valore da assegnare alla variabile stessa. Nel riferimento dell'esempio precedente, i costruttori per la classe `stack` possono essere riscritti nella forma:

```

stack::stack(int s) : TOS(0), size(s), buffer(new data[s])
{
}

stack::stack(int s, data v) : TOS(0), size(s), buffer(new data[s])
{
    for(count=0;count<s;count++)
        buffer[count]=v;
}

```

3.2.6 Riferimento

I membri (attributi e metodi) di un oggetto possono essere referenziati in modo analogo a quanto avviene nel riferimento ai membri di una struttura, a partire da un riferimento all'oggetto nella forma `<var>.member_name` oppure a partire da una espressione che restituisce l'indirizzo dell'oggetto stesso nella forma `<expr_addr>->member_name`.

Nell'esempio che segue, il metodo `push()` viene invocato in modo equivalente a partire da un riferimento all'oggetto (`S.push(13.)`) oppure a partire da una espressione che restituisce l'indirizzo dell'oggetto stesso (`Sptr->push(13.)`):

```
void main(void)
{   stack S;
    stack *Sptr;
    ...
    Sptr=&S;
    ...
    S.push(13.);
    ...
    Sptr->push(13.);
    ...
}
```

Vale la pena di richiamare che il riferimento ai membri di una classe è soggetto alle restrizioni di visibilità dei membri nelle clausole `protected` e `public`, come descritto nella sezione 3.2.2. Nel caso dell'esempio, i due riferimenti al metodo `push()` sono legali essendo `push()` parte della clausola `public`, mentre non sarebbe stato legale un riferimento all'attributo `TOS` che è invece nella clausola `protected`.

3.2.7 Self-reference

Un oggetto può esprimere un riferimento a sé stesso con il costrutto `this`.

Dal punto di vista concettuale, è utile rimarcare che l'uso di tale costrutto è in effetti stteso tutte le volte in cui in un metodo si esprime un riferimento ad un attributo nel contesto dell'oggetto. Ad esempio, l'implementazione del metodo `push()` nella classe `stack` riportata nella sezione 3.2.1, deve essere riguardata come una contrazione sulla forma seguente:

```
boolean stack::push(data value)
{
    if(this->TOS==this->size)
        return FALSE;
    else
        {   this->buffer[this->TOS]=value;
```

```

        this->TOS++;
        return TRUE;
    }
}

```

A parte tale uso di principio, il costrutto `this` diventa essenziale quando un oggetto deve passare ad un altro il proprio riferimento. Un classico schema in cui questo avviene è la *callback*: un oggetto A passa il proprio riferimento che poi verrà usato da un qualche altro oggetto B per invocare un metodo su A stesso. Tale schema viene spesso usato per realizzare schemi di *inversione della responsabilità* nella quale l'oggetto A invece che chiamare B per effettuare una qualche operazione lascia che sia B ad assumere l'iniziativa di chiamarlo.

Il meccanismo è illustrato nell'esempio che segue. Le classi `subject` e `observer` hanno ciascuna un attributo privato `state`, e i valori dei due devono rimanere allineati. Per questo il `subject` conosce l'indirizzo dell'`observer` e il metodo `subject::set_state()` che modifica lo `state` include una invocazione al metodo `observer::update()` con il quale `subject` passa ad `observer` il proprio indirizzo. In base ad una qualche condizione non precisa nell'esempio, `observer` decide se allineare il proprio stato a quello di `subject`; in caso positivo, usa l'indirizzo ricevuto per invocare `subject::get_state()` e farsi restituire il valore aggiornato dello stato.

```

class observer;
class subject;

class observer
{
private:
    int state;
public:
    void update(subject * S_ptr);
    int get_state(void);
};

class subject
{
private:
    int state;
    observer * observer_ptr;
public:
    subject(observer * o_ptr);
    void set_state(int value);
    int get_state(void);
};

int main(void)

```

```

{
    subject * s_ptr;
    observer * o_ptr;
    o_ptr=new observer;
    s_ptr = new subject(o_ptr);
    ...
    s_ptr->set_state(10);
    printf("\n subject.state=%d",s_ptr->get_state());
    printf("\n observer.state=%d",o_ptr->get_state());
    ...
}

subject::subject(observer * o_ptr)
{
    observer_ptr=o_ptr;
}

void subject::set_state(int value)
{
    state=value;
    observer_ptr->update(this);
}

int subject::get_state(void)
{
    return state;
}

void observer::update(subject * s_ptr)
{
    if(...)
    {   state=s_ptr->get_state();
    }
}

int observer::get_state(void)
{
    return state;
}

```

3.3 Attributi e metodi statici

In C una variabile può essere associata a *storage class* di tipo *static*. Questo le attribuisce tempo di vita globale e fa sì che essa sia inizializzata solo alla prima attivazione della funzione che la contiene, pur mantenendone la visibilità limitata al corpo della funzione nella quale è dichiarata. Questo costituisce un *idioma*

utile a mantenere memoria tra successive chiamate di una stessa funzione, come illustrato nell'esempio che segue:

```
void only_once(void)
{
    static int first_call=0;

    if(first_call==0)
    {
        printf("\n first call! ");
        first_call=1;
    }else
        printf("\n subsequent call");
}
```

La variabile `first_call` viene inizializzata a 0 solo alla prima esecuzione della funzione `only_once()`. Quando la funzione restituisce il controllo, il valore che ha attribuito alla variabile viene conservato e recuperato nella successiva chiamata. Nell'esempio, questo è usato per fare sì che la funzione stampi il messaggio `first call` solo in occasione della prima chiamata. Uno schema del tutto analogo potrebbe servire a realizzare una funzione che mantiene traccia del numero di volte che viene attivata.

Il concetto di *membro statico* di una classe c++ estende il concetto di variabile statica del c applicandolo non solo agli attributi ma anche ai metodi.

Un attributo statico è condiviso tra tutti gli oggetti istanziati nella classe ed ha tempo di vita globale, precedente alla creazione del primo oggetto nella classe stessa. Questo permette di realizzare *schemi* nei quali le diverse istanze di una classe concorrono alla manipolazione di una variabile tra loro condivisa.

Anche un metodo può essere associato allo storage class static. Così facendo il metodo assume visibilità globale e può essere invocato prima della creazione di alcuna istanza della classe stessa.

Nell'esempio che segue, la classe `counted_class` include un attributo statico `instances` che realizza un contatore del numero di istanze nella classe. Essendo static, la variable `instances`, è allocata staticamente e inizializzata a 0 prima della creazione di alcuna istanza nella classe; alla creazione di ogni istanza nella classe, il costruttore incrementa la variabile; viceversa a ogni distruzione la variabile viene decrementata. Il metodo statico `get_instances_number()` restituisce il numero di istanze della classe. È globalmente visibile e accessibile in qualsiasi altra classe. La particolarità è che esso può essere invocato anche nel caso in cui della classe non siano ancora state create istanze (indicando il nome della classe a prefisso del nome del metodo, i.e. con una espressione del tipo: `x=counted_class::get_instances_number();`):

```
class counted_class
```

```

{ private:
    static int instances=0;
    ...
public:
    counted_class(void);
    ~counted_class(void);
    static int get_instances_number();
    ...
};

counted_class::counted_class(void)
{ ...
    instances++;
}

counted_class::~counted_class(void)
{ ...
    instances--;
}

int counted_class::get_instances_number(void)
{
    return instances;
}

```

3.4 Ereditarietà

Il C++ permette di definire una classe come *derivazione* di una classe *base*. Questo realizza un meccanismo di *ereditarietà* nel senso che: *i)* la classe derivata può fare riferimento ad attributi e metodi non privati della classe base come se questi fossero parte della classe derivata stessa; *ii)* può esporre attributi e metodi pubblici della classe base nell'interfaccia offerta a clienti esterni; e *iii)* può a sua volta trasferirli in eredità ad ulteriori classi derivate.

3.4.1 Ereditarietà di implementazione

Il meccanismo dell'ereditarietà permette di *estendere* una classe di oggetti secondo un principio per cui una o più classi derivate *specializzano* il comportamento di una classe base, aggiungendo attributi e metodi che realizzano la specializzazione senza dovere replicare quanto già definito nella classe base.

L'ereditarietà permette di condividere codice tra diverse specializzazioni di una stessa classe e abilita un principio di evoluzione del codice noto come *open/close*: una classe derivata può modificare una classe base per adattarla a nuovi requisiti (in questo senso il codice è *open*) senza però modificare la classe preesistente

pregiudicandone la compatibilità con preesistente codice (in questo senso il codice è *close*).

Illustriamo il concetto considerando il caso nel quale si voglia realizzare una classe che riproduce lo stato e il comportamento di uno stack ma vi aggiunge la capacità di accedere ai valori anche in posizioni diverse dalla ultima e comunque senza rimuoverli dallo stack stesso. L'estensione potrebbe essere ottenuta modificando la definizione della classe stack che già abbiamo definito, aggiungendo un indice che codifica una posizione corrente entro il buffer e alcune funzioni che permettono la lettura del valore alla posizione corrente e lo spostamento della posizione corrente. Così facendo la modifica potrebbe però avere impatto su eventuali classi che già esistono e fanno riferimento alla versione esistente della classe stack. Questo approccio non sarebbe comunque conveniente nel caso in cui si dovessero realizzare più specializzazioni adattate a diversi contesti.

Il problema può essere evitato definendo una classe *visitable_stack* che eredita le definizioni di *stack* e vi aggiunge i soli attributi e metodi che implementano le nuove responsabilità. L'uso di *stack* come classe base è specificato nella riga iniziale di definizione della classe:

```
class visitable_stack: stack
{
protected:
    int current;

public:
    Boolean forward(void);
    Boolean backward(void);
    Boolean gotoTOS(void);
    Boolean get_value(data & value);
};

Boolean visitable_stack::forward(void)
/* se la posizione corrente non e' l'ultima, la avanza e restituisce TRUE
   altrimenti restituisce FALSE */
{
    if(current<TOS)
    {
        current++;
        return TRUE;
    }else
        return FALSE;
}

Boolean visitable_stack::get_value(void)
/* se lo stack non e' vuoto restituisce il valore alla posizione corrente
   e restituisce TRUE, altrimenti restituisce FALSE */
{
    if(TOS!=0)
```

```

    {   value=buffer[current];
        return TRUE;
    }else
        return FALSE;
}

```

Come mostrato nell'esempio, all'interno della classe derivata è possibile fare riferimento ai membri `public` e `protected` della classe base (e.g. `TOS` e `buffer`). Analogamente, una funzione che dispone di un riferimento ad un oggetto della classe derivata può invocare indifferentemente i membri pubblici dichiarati nella classe derivata (e.g. `push()`) o i membri pubblici ereditati dalla classe base (e.g. `get_value()`):

```

void client(void)
{
    visitable_stack S(10);
    data value;
    Boolean data_available;
    ...
    S.push(15.);
    ...
    data_available = S.get_value(value);
    if(data_available==TRUE)
        cout << value ;
    ...
}

```

3.4.2 Ereditarietà in modo `public`, `protected` e `private`

Nella definizione di una classe derivata è possibile specificare uno di tre diversi modi di eredità, premettendo il qualificatore `public`, `protected` o `private` al nome della classe base nella definizione della classe derivata.

Nell'esempio del paragrafo precedente, non era specificato alcuno dei tre modi. Per default questo equivale ad assumere il modo `public`, che potrebbe essere esplicitamente indicato modificando la testata della definizione della classe nella forma:

```

class visitable_stack: public stack
{ ... };

```

Il modo di eredità determina quali membri della classe base possono essere riferiti nella classe derivata, e con quale livello di visibilità essi figurano nell'interfaccia della classe derivata verso clienti esterni o verso ulteriori derivazioni:

- nella derivazione in modo `public`, la classe derivata eredita i membri `public` e `protected` della classe base mantenendone invariato il livello di visibilità;

- nella derivazione in modo `protected`, la classe derivata eredita i membri `public` e `protected` della classe base ma li espone come `protected`;
- nella derivazione in modo `private`, la classe derivata eredita i membri `public` e `protected` della classe base ma li espone come `private`;

3.4.3 Ereditarietà dei costruttori

Alla creazione di un oggetto che appartiene ad una classe derivata, il costruttore della classe base è invocato prima del costruttore della classe derivata. Se la classe derivata non definisce alcun costruttore allora il suo costruttore di default invoca il costruttore `void` della classe base, purchè nella classe base sia definito esplicitamente un costruttore `void` oppure non sia definito alcun costruttore. Nel caso in cui la classe base abbia più costruttori, il costruttore della classe derivata può specificare quale invocare.

Nell'esempio che segue, viene definito per la classe `visitable_stack` un costruttore che riceve come unico parametro la dimensione dello stack da allocare. Esso invoca il costruttore della classe base che riceve due parametri (la dimensione dello stack e il valore di inizializzazione in questo caso nullo), e poi provvede a dare valore all'attributo `current`:

```
visitable_stack::visitable_stack(int size):stack(size,0)
{
    current=0;
}
```

3.5 Polimorfismo e dynamic binding

3.5.1 Override e polimorfismo

Una classe derivata può ridefinire (*override*) un metodo ereditato da una classe base. Per questo è sufficiente che la classe derivata includa la dichiarazione e anche l'implementazione. In questo caso si dice che il metodo ha una implementazione *polimorfica* in quanto la sua invocazione risulta nell'esecuzione di un diverso codice a seconda che l'operazione sia applicata ad un oggetto istanziato nella classe base o nella classe derivata.

Si osservi che questo è solo in apparenza simile al meccanismo dell'*overloading* menzionato nel capitolo 3.2.3: mentre nell'overloading una stessa classe può includere più funzioni con lo stesso nome ma parametri diversi nel numero o nel tipo, nell'override si hanno implementazioni della stessa funzione con gli stessi parametri all'interno di una gerarchia di ereditarietà.

Il metodo ridefinito spesso è implementato *aggiungendo* del comportamento in accordo con la specializzazione concettuale della classe derivata, prima e/o dopo il comportamento già implementato nell'omonimo metodo della classe base. Per questo, l'implementazione nella classe derivata ha necessità di fare riferimento alla implementazione nella classe base, il che può essere ottenuto con una *upcall* nella quale viene indicato esplicitamente il nome della classe base a prefisso del nome del metodo.

L'esempio seguente illustra il concetto in riferimento al caso di una classe *set* che rappresenta un insieme interi su una lista collegata con puntatori, la quale viene estesa da una classe *counted_set* per aggiungere la capacità di mantenere il conto del numero di elementi contenuti nell'insieme. Per questo, *counted_set()* aggiunge un attributo privato *cardinality*, espone un nuovo metodo *get_cardinality()* che ne restituisce il valore, e ridefinisce il metodo *add_element()* in modo che questo, oltre a quanto già implementato nella classe base, provveda a incrementare la cardinalità. Nella classe *counted_set* viene anche ridefinito il costruttore in modo da provvedere alla inizializzazione della cardinalità. Vale la pena di osservare che il costruttore void della classe derivata deve essere riguardato come un override del costruttore void della classe base pur avendo i due, ovviamente, un nome diverso:

```
struct list {
    int value;
    struct list * next_ptr;
};

class set
{ private:
    struct list * head_ptr;
    struct list * current_ptr;
public:
    set(void);
    void add_element(int e);
    void start_iteration(void);
    bool iteration_completed(void);
    int get_current_value(void);
    void next(void);
};
set::set(void)
{   head_ptr=0;
    current_ptr=0;
}
void set::add_element(int e)
{   struct list * tmp_ptr;

    tmp_ptr=head_ptr;
    head_ptr=(struct list *)malloc(sizeof(struct list));
```

```

if(head_ptr!=NULL)
{   head_ptr->value=e;
    head_ptr->next_ptr=tmp_ptr;
}
else
{...
}
}

void set::start_iteration(void)
{   current_ptr=head_ptr;
}
int set::get_current_value(void)
{   return current_ptr->value;
}
void set::next(void)
{   current_ptr=current_ptr->next_ptr;
}
bool set::iteration_completed(void)
{   if(current_ptr==NULL)
        return TRUE;
    else
        return FALSE;
}

class counted_set: public set
{   private:
    int      cardinality;
    public:
        counted_set(void);
        void    add_element(int e);
        int     get_cardinality(void);
};
counted_set::counted_set(void): cardinality(0)
{
}
void counted_set::add_element(int e)
{   cardinality++;
    set::add_element(e);
}
int counted_set::get_cardinality(void)
{   return cardinality;
}

```

3.5.2 Dynamic binding

Il meccanismo dell'override introduce la possibilità che in una *gerarchia di ereditarietà* esistano più implementazioni di uno stesso metodo. Questo pone al compilatore un problema di *binding* che consiste nel decidere quale implementazione legare ad una chiamata.

La decisione è basata sul tipo della classe che compare nel riferimento al metodo. Nel caso di un riferimento nella forma <var>.op_name(), il tipo è quello di <var>, che necessariamente coincide con il tipo della classe nel quale è stato istanziato l'oggetto. Se però il riferimento è espresso nella forma <expr_addr>->op_name(), allora il tipo è quello di <expr_addr>, che per effetto di un *cast* potrebbe anche essere diverso dal tipo della classe nella quale è stato istanziato l'oggetto.

Una volta determinata la classe a cui è legata l'invocazione, se questa non dispone di una propria implementazione del metodo, allora viene utilizzata la prima implementazione incontrata risalendo la gerarchia di ereditarietà.

I diversi casi sono illustrati nell'esempio che segue: la classe base definisce un contatore counter come attributo protetto e due metodi op() e opv() che lo incrementano di 1; i due metodi sono equivalenti salvo che opv() è dichiarato virtuale. La classe base è *sub-classed* in una gerarchia di ereditarietà di altri 3 livelli: dbase eredita counter, op() e opv() da base; a sua volta ddbase eredita da dbase l'attributo counter ma ridefinisce op() e opv(); infine, dddbase eredita counter, op() e opv() da ddbase senza alcun override.

```
class base
{ protected:
    int counter;
public:
    void op(void);
};

void base::op(void)
{ counter++; }

class dbase: public base
{ };

class ddbase: public dbase
{ public:
    void op(void);
};

void ddbase::op(void)
{ counter*=2;
}

class dddbase: public ddbase
{ };

int main(void)
{ dddbase * ptr=new dddbase;
    ptr->op();                                // viene legata ddbase::op()
    ((dbase *) ptr)->op();                   // viene legata base::op()
```

```

    dbase * ptr2= new dbase;
    ((ddibase *) ptr2)->op();           // viene legata ddbase::op()
}

```

Nel corpo della funzione `main()`, sono istanziati due oggetti nelle classi `ddibase` e `dbase` i cui indirizzi sono memorizzati rispettivamente nei puntatori `ptr` e `ptr2`:

- nel caso `ptr->op()` il legame è determinato dal tipo di `ptr` che è un puntatore a `ddibase`; poiché `ddibase` non dispone di un'implementazione di `op()`, viene legata l'implementazione ereditata da `dbase`.
- nel caso `((dbase *) ptr)->op()` viene effettuato un *upcast* di `ptr` nel tipo puntatore a `dbase`; l'implementazione legata è quindi quella di `dbase`, ovvero quella ereditata da `base`.
- nel caso `((ddibase *) ptr2)->op();`, il *downcast* ha invece effetto per cui viene legata l'implementazione di `ddibase` che è identificata risalendo la gerarchia a partire dalla classe `ddibase` espressa nel riferimento.

3.5.3 Sostituibilità e metodi virtuali

La relazione di ereditarietà tra una classe `base` e una classe derivata sottende un concetto di *sostituibilità*: un'istanza della classe derivata può operare in qualsiasi ruolo nel quale sia attesa un'istanza della classe base. Nell'esempio del capitolo 3.5.2, un client che delega l'operazione `op()` ad un oggetto di classe `base` può delegare la stessa operazione ad un oggetto di tipo `ddibase` o `dbase`: poiché entrambe derivano da `base` attraverso una gerarchia di eredità in modo `public`, entrambe includono nella loro interfaccia tutti i metodi pubblici di `set`.

I metodi *virtuali* sono introdotti nel linguaggio per permettere al client di rimanere schermato dal conoscere quale forma concreta viene sostituita all'interfaccia che esso osserva.

Uno o più metodi di una classe possono essere dichiarati *virtuali*. Questo si ottiene premettendo la direttiva `virtual` al prototipo del metodo nella definizione della classe. Ad esempio, in riferimento al caso del capitolo 3.5.2, il metodo `op()` è dichiarato virtuale modificando la definizione della classe `base` come segue:

```

class base
{
protected:
    int counter;
public:
    virtual void op(void);
};

```

Un metodo dichiarato virtuale in una classe base rimane virtuale nell'ambito di tutta la gerarchia di ereditarietà che estende la classe stessa, anche se la dichiarazione virtual non è ripetuta nelle classi derivate. Nel riferimento dell'esempio, la dichiarazione virtual nella classe base si applica anche alle classi derivate dbase, ddbase e dddbase che estendono successivamente base.

La dichiarazione virtual modifica la regola di binding applicata al metodo: per un metodo dichiarato virtuale viene sempre invocata l'implementazione della classe nel cui tipo è stato istanziato l'oggetto, indipendentemente dal tipo della classe che compare nel riferimento. Il seguente frammento di codice illustra il concetto, facendo ancora riferimento alla gerarchia di ereditarietà da base a dddbase precedentemente introdotta:

```
int main(void)
{   base * ptr;
    ...
    if(...)
        ptr=new ddbase;
    else
        ptr=new base;
    ...
    ptr->op();           // chiama base::op() oppure ddbase::op()
                        // a seconda del tipo su cui e' stato
                        // istanziato l'oggetto puntato da ptr
}
```

Nell'esempio, il `main()` istanzia un oggetto nella classe `base` o nella classe `ddbase` a seconda di una qualche condizione non rappresentata. In entrambi i casi, il puntatore all'oggetto istanziato viene memorizzato in un puntatore a `base` denominato `ptr`. Quando successivamente `op()` viene invocato con il riferimento `ptr->op()`, l'implementazione legata è quella del tipo nel quale l'oggetto puntato da `ptr` è stato concretamente istanziato.

Così facendo il `main()` localizza la scelta dell'implementazione concreta di `base` utilizzata al solo frammento di codice iniziale, evitando di dovere ripetere in modo consistente una stessa clausola `if/else` ad ogni successivo uso dell'oggetto puntato da `ptr`.

Il disaccoppiamento può essere ulteriormente perfezionato delegando ad una classe `factory` la responsabilità di istanziare un oggetto nella classe `base` o `ddbase` secondo una decisione presa in accordo ad una qualche informazione di contesto (e.g. un numero di versione dell'applicazione). In questo caso il `main()` rimane completamente *unaware* della forma concreta nella quale è istanziato l'oggetto che implementa l'interfaccia di `base` che esso attende:

```
class factory
```

```

{ private:
    data context;
public:
    base* get_base(void);
};

base * factory::get_set(void)
{ if(context==...)
{   return new base;
} else
{   return new ddbase;
}
}

void main(void)
{ base * ptr;
factory * factory_ptr;
...
ptr=factory_ptr->get_base();
...
ptr->op();
...
}

```

3.5.4 Il problema della classe di base fragile

L'uso di metodi virtuali permette di ridurre l'accoppiamento tra un client e la gerarchia di ereditarietà sviluppata a partire da una classe base, aumentando fortemente l'efficacia del principio di manutenzione *open/close* menzionato nel capitolo 3.4.1. Tuttavia, l'astrazione che i metodi virtuali inducono può risultare in comportamenti inattesi, laddove la programmazione non sia sostenuta da adeguati strumenti di visione dell'interazione tra le classi. Illustriamo il concetto facendo riferimento ad un classico difetto nella progettazione ad oggetti noto come *fragile base class*.

L'esempio estende il caso delle classi `set` e `counted_set` introdotte nel capitolo 3.5.1: supponiamo che oltre al metodo `add_element()` la classe base `set()` includa anche un metodo `join_set()` il quale unisce all'insieme rappresentato tutti gli elementi di un altro insieme usando ripetutamente `add_element()`; supponiamo anche che nella classe derivata `counted_set()` sia definito un metodo `join_counted_set()` che unisce un `counted_set()`; supponiamo infine che almeno il metodo `add_element()` nella classe `set` sia dichiarato virtuale. Riportiamo del codice le parti modificate e quelle aggiunte:

```

class set
{
...
public:
    virtual void    add_element(int e);

```

```

        void      join_set(set* set_ptr);
        ...
};

void set::join_set(set* set_ptr)
{   set_ptr->start_iteration();
    while(set_ptr->iteration_completed()==FALSE)
    {   add_element(set_ptr->get_current_value());
        set_ptr->next();
    }
}

class counted_set: public set
{ private:
    int cardinality;
public:
    counted_set(void);
    void    add_element(int e);
    int     get_cardinality(void);
    void    join_counted_set(counted_set* cset_ptr);
};
void counted_set::join_counted_set(counted_set* cset_ptr)
{   cardinality+=cset_ptr->get_cardinality();
    join_set(cset_ptr);
}

int main(void)
{   counted_set set1;
    counted_set set2;
    ...
    set2.join_counted_set(&set1);
    ...
}

```

L'invocazione di `join_counted_set()` nel `main()` produce un malfunzionamento: `join_counted_set()` incrementa `set2.cardinality` del valore contenuto in `set1.cardinality` e poi esegue il metodo `join_set()` ereditato dalla classe base `set`; in tale metodo viene ripetutamente invocato `add_element()`, il quale è però virtuale e viene quindi legato all'implementazione della classe `counted_set`; in tale implementazione, per ogni elemento del set da unire, viene incrementata `set2.cardinality` e poi viene aggiunto l'elemento usando il metodo della classe base `set` con una *upcall* esplicita. Così facendo, al termine dell'operazione, `set2.cardinality` è stata incrementata del doppio del valore di `set1.cardinality`.

Il problema è evidentemente legato al fatto che nella classe base `set` esiste un metodo pubblico (`join_set()`) che usa un metodo virtuale (`add_element()`). In tale condizione, l'override di `add_element()` *rompe l'incapsulamento* della classe base `set` in quanto modifica l'implementazione del metodo `join_set()` sosti-

tuendo la delega al metodo `set::add_element()` con la delega ad un metodo `counted_set::add_element()` che non poteva evidentemente essere previsto al momento della realizzazione della classe `set`.

Un possibile modo per evitare la fragilità è quello di portare il codice di `add_element()` in un metodo privato che denominiamo `priv_add_element()`, e utilizzare tale metodo sia nell'implementazione di `add_element()` che in quella di `join_set()`:

```
class set
{ private
    void    priv_add_element(int e);
    ...
public:
    virtual void    add_element(int e);
    void    join_set(set* set_ptr);
    ...
};

void set::priv_add_element(int e)
{
    struct list * tmp_ptr;

    tmp_ptr=head_ptr;
    head_ptr=(struct list *)malloc(sizeof(struct list));
    if(head_ptr!=NULL)
    {
        head_ptr->value=e;
        head_ptr->next_ptr=tmp_ptr;
    }
    else
    {...}
}

void set::add_element(int e)
{
    priv_add_element(e);
}

void set::join_set(set* set_ptr)
{
    set_ptr->start_iteration();
    while(set_ptr->iteration_completed()==FALSE)
    {
        priv_add_element(set_ptr->get_current_value());
        set_ptr->next();
    }
}
```

3.5.5 Classi astratte e interfacce

Una classe può omettere la definizione di un metodo dichiarato virtuale, rimandandola ad una classe derivata. Questo si ottiene "inizializzando a 0" la dichiarazione del metodo nella definizione della classe e omettendo poi la definizione del metodo

stesso nell'implementazione della classe. Ad esempio, per il metodo `op()` nella classe base definita nel capitolo 3.5.1:

```
class base
{ protected:
    int counter;
public:
    virtual void op(void)=0;
};
```

Una classe che include almeno un metodo virtuale non implementato si dice *astratta*. Evidentemente essa non può essere istanziata e serve invece come base sulla quale è possibile definire classi estese le quali forniranno diverse implementazioni dei metodi omessi riflettendo diverse specializzazioni. L'uso di classi astratte costituisce un fondamentale meccanismo di disaccoppiamento, che permette ad un client di dipendere dall'interfaccia esposta su una classe astratta senza conoscere i dettagli sul modo con cui l'interfaccia è concretamente implementata.

Una classe si dice *astratta pura* quando essa non contiene la definizione di alcuno dei suoi metodi. L'estensione di una classe di tale tipo evidentemente non serve a realizzare ereditarietà di implementazione ma piuttosto ad abilitare il meccanismo della sostituibilità.

Illustriamo il concetto nel riferimento di uno schema ampiamente usato nella buona pratica della programmazione ad oggetti (e.g. negli schemi *bridge* o *strategy*): un `client()` utilizza i metodi di un'interfaccia; tali metodi sono esposti su una classe astratta pura e possono essere realizzati diversamente in una varietà di classi concrete diverse; il `client` è disaccoppiato dalla scelta della particolare implementazione concreta attraverso una classe `factory` che assume la responsabilità di decidere quale forma concreta istanziare in base ad una qualche condizione che caratterizza il contesto.

L'esempio *rifactorizza* la classe `set` già definita nel capitolo 3.5.1 ridenominandola `set_on_list` e definendola come implementazione di una superclasse astratta `set`. Questo apre la via alla definizione di una implementazione alternativa, ad esempio basata su un albero binario anziché su una lista collegata con puntatori:

```
class set
{ public:
    virtual void add_element(int e)=0;
    virtual void join_set(set* set_ptr)=0;
    virtual void start_iteration(void)=0;
    virtual bool iteration_completed(void)=0;
    virtual int get_current_value(void)=0;
    virtual void next(void)=0;
};
```

```

class set_on_list: public set
{ private:
    struct list * head_ptr;
    struct list * current_ptr;
    void priv_add_element(int e);
public:
    set(void);
    void add_element(int e);
    void join_set(set* set_ptr);
    void start_iteration(void);
    bool iteration_completed(void);
    int get_current_value(void);
    void next(void);
};

class set_on_tree: public set
{ private:
    struct btree * root_ptr;
    struct btree * current_ptr;
    void priv_add_element(int e);
public:
    set(void);
    void add_element(int e);
    void join_set(set* set_ptr);
    void start_iteration(void);
    bool iteration_completed(void);
    int get_current_value(void);
    void next(void);
};

```

Si osservi che un cliente della classe `set` risulta a questo punto disaccoppiato dalla conoscenza del modo concreto con cui viene implementato il tipo. Si osservi anche che in questo modo diventa possibile combinare in modo del tutto trasparente insiemi rappresentati su liste con insiemi rappresentati su alberi binari. Lo stesso concetto potrebbe essere applicato per rendere trasparente la distinzione tra un insieme di interi e un insieme di float.

3.5.6 Ereditarietà multipla

Una classe può derivare contemporaneamente da più classi base. Dal punto di vista sintattico questo si ottiene elencando tutte le classi base sulla riga di apertura della definizione di una classe.

L'esempio seguente illustra la sintassi e il concetto facendo riferimento ad uno schema noto come *adapter* che serve ad adattare una implementazione esistente ad una nuova interfaccia attesa. Nell'esempio: la classe `adaptee` rappresenta un

punto in coordinate polari `alpha` e `radius`; viceversa nel corpo di `main()` si vuole usare una rappresentazione in coordinate Cartesiane `x` e `y`.

Per adattare l'interfaccia esposta da `adaptee` alle attese del `main()` viene definita una classe astratta pura `target` che espone l'interfaccia attesa da `main()`. Si definisce poi una classe `adapter` la quale eredita sia da `target` che da `adaptee`: la classe `adapter` utilizza i metodi ereditati da `adaptee` per implementare i metodi virtuali prescritti da `target`. Questo costituisce un uso combinato di ereditarietà di implementazione (quella per cui `adapter` può usare i metodi di `adaptee`) e ereditarietà di interfaccia (per cui un oggetto di tipo `adapter` può sostituirsi nel `main()` ad un oggetto di tipo `target`).

Si osservi che l'ereditarietà è pubblica rispetto all'interfaccia astratta `target` ma privata rispetto all'oggetto concreto `adaptee`: questo concede ad `adapter` un *minimo privilegio* nell'accesso ad `adaptee` in modo da impedire ad un client esterno di invocare su `adapter` i metodi ereditati da `adaptee`.

```
class adaptee
{ private:
    float alpha;
    float radius;
public:
    float get_alpha(void);
    float get_radius(void);
};

float adaptee::get_alpha(void)
{   return alpha;
}

float adaptee::get_radius(void)
{   return radius;
}

class target
{ public:
    virtual float get_x(void)=0;
    virtual float get_y(void)=0;
};

class adapter:private adaptee, public target
{
    float get_x(void);
    float get_y(void);
};

float adapter::get_x(void)
{   return adaptee::get_radius()*cos(adaptee::get_alpha());}

float adapter::get_y(void)
{   return adaptee::get_radius()*sin(adaptee::get_alpha());}
```

3.6 Ulteriori letture

La conoscenza del c++ può essere utilmente estesa in almeno tre diversi aspetti.

- Un primo aspetto è il completamento della conoscenza sintattica e semantica del linguaggio. Che include un numero di sottigliezze e effetti collaterali, non fondamentali in una programmazione anche avanzata, ma comunque non facili da inquadrare in un modello compatto e maneggevole. Il problema della fragile base class illustra bene il concetto.

Nella mia esperienza, non è questa la maggiore difficoltà né la direzione nella quale è interessante spingere l'apprendimento. Esistono comunque una varietà di testi che forniscono una trattazione estensiva, e fra questi il testo di H.Schildt citato in bibliografia è un buon riferimento.

- Un secondo aspetto è l'acquisizione di uno strumento di astrazione che permetta di inquadrare al giusto livello il progetto delle classi, delle loro responsabilità e delle loro relazioni.

Lo strumento definitivamente consolidato per questo è lo Unified Modeling Language (UML). In particolare, è circa impossibile produrre codice di valore senza usare in esplicito o a mente il diagramma delle classi (class diagram). Di minore necessità ma talvolta utili risultano anche i diagrammi di sequenza o di collaborazione (sequence e collaboration diagrams). I diagrammi dei casi d'uso (use case diagrams) e il loro corrispondente testuale riguardano il problema della modellazione dei requisiti più della progettazione delle classi ma mi risulta difficile non averli almeno una volta menzionati parlando di UML. Cosiccome è utile menzionare come ulteriore passo dell'apprendimento la comprensione dei modelli di sviluppo e organizzazione del ciclo di vita del SW, che in quest'epoca sono lo Unified Process (UP) e l'eXtreme Programming (XP).

Ottimi testi di riferimento per questa direzione di apprendimento sono "UML distilled" di Martin Fowler, e "Embracing change with eXtreme Programming," di Kent Beck. Per UP non esiste un riferimento altrettanto efficace, ma un discreto compromesso è raggiunto nel testo di J.Arlow, e I.Neustadt citato in bibliografia.

- Il terzo aspetto, il più complesso e divertente, è la comprensione di come le potenzialità del linguaggio sono usate nell'ambito di un approccio sistematico e guidato dal metodo. Rientrano in questo ambito problemi più astratti come: la scelta tra schemi basati sulla delega rispetto che sulla ereditarietà; l'uso sistematico delle interfacce; l'uso di polimorfismo e dynamic binding come meccanismo di disaccoppiamento e di riduzione dei conditionals nel

codice; il trade-off tra facilità di costruzione degli oggetti, la flessibilità nella loro manipolazione, e la complessità nella loro distruzione; la progettazione di strutture che catturano nel livello statico delle classi chiusure strategiche che facilitano la verificabilità, la manutenibilità e la capacità di evoluzione del codice.

Comprendere questi meccanismi fa in effetti la differenza tra conoscere il linguaggio c++ e sapere programmare, ed apre la vista su un ragionamento logico al tempo stesso affascinante e concreto.

Senza dubbio il testo più raccomandato per raggiungere questo livello della programmazione e progettazione è "Design Patterns" della cosiddetta Gang Of Four. Il libro illustra un insieme di 23 schemi, combinando una discussione elevata sulla logica e i valori della progettazione orientata agli oggetti, con schemi che mostrano la faccia utile dell'UML e frammenti di codice c++ che illustrano un uso avanzato e sobrio del linguaggio.

Testi di approfondimento

Architettura dei calcolatori:

- D.A.Patterson, J.L.Hennessy, "Struttura e progetto dei calcolatori," Zanichelli, Bologna, 1995.
- G.Bucci, "Architettura e organizzazione dei calcolatori elettronici - fondamenti ,," McGraw-Hill Italia, Milano, 2001.

Linguaggio c:

- H.Schildt, "c, the complete reference," McGraw-Hill, 2000.
- E.Yourdon, L.Constantine, "Structured design: fundamentals of a discipline of Computer programming and system design," Prentice Hall, 1986.

Linguaggio c++:

- E.Gamma, R.Helm, R.Johnson, J.Vlissides, "Design Patterns: elements of reusable object oriented software," Addison Wesley, 1995.
- M.Fowler, "UML distilled," Addison Wesley, 2000.
- H.Schildt, "c++, the complete reference," McGraw-Hill, 2002.
- J.Arlow, I.Neustadt, "UML e Unified Process," McGraw Hill, 2003.
- K.Beck, "Embracing change with eXtreme Programming," *IEEE Computer*, Vol.32, No.10, 1999.

Strutture dati, complessità e algoritmi:

- T.H.Cormen, C.E.Leiserson, R.L.Rivest, "Introduction to Algorithms," MIT Press, 1990.
- R.Sedgewick, "Algorithms in C" Addison Wesley, 1990.
- R.Sedgewick, "Algorithms in C++, parts 1-4" Addison Wesley, 1998.
- C.Batini, L.Carlucci Aiello, M.Lenzerini, A.Marchetti Spaccamela, A.Miola, "Fondamenti di Programmazione dei Calcolatori Elettronici," Franco Angeli Editore Milano, 1993.

