Azzolini Riccardo 2021-03-17

Preprocessore

1 Preprocessore

In un file di codice C, ogni riga che inizia con il carattere `#` non fa tecnicamente parte del linguaggio C, ma è invece una **direttiva al preprocessore**. Infatti, complessivamente, il *sistema C* è costituito da:

- il compilatore, che implementa il linguaggio C "vero e proprio";
- il **preprocessore**, che esegue delle trasformazioni sul codice sorgente prima che questo venga compilato.

In precedenza è già stata stata spiegata la direttiva `#include`. Adesso, si vedranno le altre principali direttive.

2 Macro

La direttiva `define` permette di definire delle **macro**, dei frammenti di codice sorgente riutilizzabili associati a dei nomi.

Una macro senza parametri viene definita con la sintassi

`#define` *`nome testo_di_sostituzione`*

dove *`nome`* è un identificatore e *`testo_di_sostituzione`* è un qualunque frammento di testo. L'effetto di questa direttiva è che, a partire dal punto del codice in cui essa compare, il preprocessore sostituisce letteralmente ogni occorrenza dell'identificatore *`nome`* con il *`testo_di_sostituzione`*. È importante sottolineare che questa sostituzione avviene in modo *letterale*, praticamente senza considerare le regole del linguaggio C — di fatto, corrisponde all'operazione "search and replace" di un editor di testo. Si vedrà a breve che ciò ha importanti conseguenze su come le macro devono essere definite e usate.

È anche possibile definire macro con parametri, tramite la sintassi

`#define` *`nome(parametri_formali) testo_di_sostituzione`*

dove *`parametri_formali`* è una lista di zero o più identificatori, separati da virgole, che costituiscono i parametri formali della macro. Tra il nome della macro e la parentesi

della lista dei parametri formali *non* devono esserci spazi, altrimenti la direttiva verrebbe interpretata come la definizione di una macro senza parametri.

Una macro con parametri viene utilizzata con la sintassi

nome(*parametri_attuali*)

dove *parametri_attuali* è una lista di frammenti di codice arbitrari, separati da virgole, che devono essere tanti quanti sono i parametri formali della macro. A partire dal punto in cui la macro è definita, il preprocessore sostituisce ogni occorrenza di questa sintassi con il *testo_di_sostituzione*, nel quale eventuali occorrenze dei parametri formali vengono a loro volta sostituite con i corrispondenti parametri attuali.

2.1 Esempio

Si consideri un file sorgente C contenente il seguente codice:

```
#define DIM_BUFFER 100
#define SCAMBIA(X, Y, T) T = X; X = Y; Y = T

int A[DIM_BUFFER];

int main(int argc, const char *argv[]) {
    int a = 1;
    int b = 2;
    int temp;
    SCAMBIA(a, b, temp);
    // ...
}
```

Quando si compila questo file, viene prima eseguito il preprocessore, che modifica l'input del compilatore sostituendo le occorrenze di `DIM_BUFFER` e `SCAMBIA` con i corrispondenti testi di sostituzione. Il testo risultante dall'esecuzione del preprocessore è il seguente:

```
int A[100];

int main(int argc, const char *argv[]) {
    int a = 1;
    int b = 2;
    int temp;
    temp = a; a = b; b = temp;
    // ...
}
```

2.2 Problema: precedenza degli operatori

Si supponga di voler definire una macro con parametri `MIN(x, y)` che restituisca il minimo tra i valori delle due espressioni passate come parametri attuali. Un primo tentativo di definizione potrebbe essere questo:

```
#define MIN(x, y) x < y ? x : y
```

Nei casi più semplici, questa definizione funziona; ad esempio, il codice

```
int a = 10, b = 200;
int c = MIN(a, b);
printf("%d\n", c);
```

viene trasformato dal preprocessore in

```
int a = 10, b = 200;
int c = a < b ? a : b;
printf("%d\n", c);
```

che al momento dell'esecuzione stamperà correttamente il valore `10` (il minimo tra i valori di `a` e `b`).

Invece, se si provasse a usare la macro `MIN` all'interno di un'espressione più complessa si otterrebbero risultati inaspettati. Ad esempio, intuitivamente ci si aspetterebbe che il frammento di codice

```
int a = 10, b = 200;
int c = 400 * MIN(a, b);
printf("%d\n", c);
```

stampi il valore `4000` ($400 \cdot$ `MIN(a, b)` $= 400 \cdot 10$), ma invece esso stampa `200`, perché la sostituzione della macro `MIN` produce il codice

```
int a = 10, b = 200;
int c = 400 * a < b ? a : b;
printf("%d\n", c);
```

nel quale l'espressione `400 * a < b ? a : b` viene interpretata come `(400 * a) < b ? a : b`, dato che l'operatore `*` ha precedenza maggiore di `<` e `?:`.

In sintesi, il fenomeno appena osservato è dovuto al fatto che le precedenze degli operatori sono applicate solo *dopo* l'operazione del preprocessore. Allora, quando si definisce una macro da usare nelle espressioni, per poterla utilizzare senza problemi in qualunque contesto è necessario racchiudere il testo di sostituzione tra parentesi:

```
#define MIN(x, y) (x < y ? x : y)
```

Così, il codice

```
int a = 10, b = 200;
int c = 400 * MIN(a, b);
printf("%d\n", c);
```

viene trasformato in

```
int a = 10, b = 200;
int c = 400 * (a < b ? a : b);
printf("%d\n", c);
```

che stampa correttamente il valore `4000`.

Tuttavia, il problema non è ancora del tutto risolto. Per illustrare altri casi problematici, è utile considerare la definizione di un'altra macro, `SQUARE(x)`, che restituisce il quadrato del valore dell'espressione passata come parametro attuale:

```
#define SQUARE(x) (x * x)
```

Siccome il testo di sostituzione è già racchiuso tra parentesi, non c'è il rischio che alcuni operandi dell'espressione `x * x` possano essere "catturati" dagli operatori di precedenza maggiore presenti nell'espressione in cui `SQUARE` viene usata. Al contrario, però, l'operatore `*` potrebbe catturare gli operandi di eventuali operatori di precedenza minore presenti nell'espressione passata come parametro attuale. Ad esempio, il frammento di codice

```
printf("%d\n", SQUARE(10 + 20));
```

stampa il valore `230`, e non `900` (30^2) come ci si potrebbe immaginare, perché il testo risultante dalla sostituzione della macro è

```
printf("%d\n", 10 + 20 * 10 + 20);
```

dove l'espressione viene interpretata come `10 + (20 * 10) + 20` a causa delle precedenze degli operatori coinvolti.

Anche in questo caso, per fare sì che il parametro attuale corrispondente a `x` sia trattato come un tutt'uno nell'espressione `x * x`, bisogna racchiudere ciascuna occorrenza del parametro formale `x` tra parentesi all'interno della definizione della macro:

```
#define SQUARE(x) ((x) * (x))
```

In tal modo, il codice

```
printf("%d\n", SQUARE(10 + 20));
```

viene trasformato dal preprocessore in

```
printf("%d\n", (10 + 20) * (10 + 20));
```

che stampa correttamente il valore `900`.

In generale, quando si definiscono le macro è bene abbondare con le parentesi, anche se non si pensa che ci possano essere problemi. Ad esempio, gli operatori `<` e `?:` usati nella definizione di `MIN` hanno precedenze molto basse, quindi è improbabile che catturino operandi delle espressioni passate come parametri, ma è bene racchiudere comunque ciascun parametro formale tra parentesi (oltre a racchiudere tra parentesi l'intera definizione, come già fatto prima):

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

2.3 Problema: espressioni con side effect

A differenza dei parametri attuali di una funzione, che vengono valutati una volta sola al momento della chiamata, i parametri attuali di una macro vengono *inseriti letteralmente zero o più volte* nel testo di sostituzione, dopodiché ciascun'istanza di ciascun parametro attuale viene valutata separatamente al momento dell'esecuzione del codice. Ciò è problematico se i parametri attuali sono espressioni che hanno dei side effect, cioè che contengono gli operatori di incremento (`++`), decremento (`--`), assegnamento (`=`, `+=`, ecc.) e/o delle chiamate a funzioni che hanno a loro volta dei side effect.

Ad esempio, si consideri ancora la macro `MIN`:

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))
```

Questa definizione ha tutte le parentesi necessarie per evitare problemi legati alle precedenze degli operatori, ma dà comunque un risultato aspettato nel seguente caso:

```
int a = 10, b = 200;
int c = MIN(++a, b);
printf("%d\n", c);
```

Il risultato che ci si potrebbe aspettare, ragionando come per le funzioni, è `11`, perché `++a` incrementa `a` e restituisce il valore incrementato, che è appunto `11`, ed è minore del valore `200` di `b`. Invece, il valore stampato è `12`, perché il risultato della sostituzione della macro è

```
int a = 10, b = 200;
int c = ++a < b ? ++a : b;
printf("%d\n", c);
```

quindi `a` viene incrementata due volte, e il valore restituito è quello ottenuto dopo il secondo incremento.

A differenza del problema delle precedenze, quello dei side effect è *ineliminabile*: non c'è un modo di modificare la definizione della macro per risolverlo. L'unica soluzione è fare molta attenzione ai side effect all'interno delle espressioni passate come parametri alle

macro, cercando se possibile di evitarli completamente. Perciò, è anche importante non confondere le macro con le funzioni "vere e proprie".

2.4 Confronto tra macro con parametri e funzioni

Una macro con parametri può sostanzialmente essere usata come una funzione. Il principale vantaggio di una macro rispetto a una funzione è che l'uso di una macro non comporta i costi (tempo e memoria) associati al meccanismo di chiamata di funzione (perché non è necessario allocare un nuovo record di attivazione, saltare alla prima istruzione della funzione e poi ritornare al chiamante al termine della funzione). D'altro canto, l'uso delle macro tende ad aumentare la dimensione del file eseguibile, dato che l'implementazione di una macro viene ricopiata ogni volta che la si usa, e ci sono tutti gli svantaggi visti prima legati al meccanismo di sostituzione letterale. Dunque, in genere, l'uso delle macro con argomenti ha senso prevalentemente per piccoli frammenti di codice che vengono utilizzati in contesti in cui le prestazioni sono critiche.[1]

3 Compilazione condizionale

Quando si scrive un programma C portabile, che deve poter essere eseguito su piattaforme (architetture hardware, sistemi operativi, ecc.) diverse, capita spesso che ci siano porzioni di codice specifiche per determinate piattaforme, le quali devono essere compilate solo quando si sta appunto producendo l'eseguibile per tali piattaforme. Un esempio di codice che va compilato solo in certi casi sono le istruzioni usate per il debugging (ad esempio, le stampe dei valori delle variabili), che non devono essere incluse nell'eseguibile distribuito agli utenti (perché producono output inutile per gli utenti, potrebbero ridurre le prestazioni, ecc.).

Per facilitare gestione di scenari come questi, il preprocessore permette di controllare quali porzioni di codice debbano essere compilate, in base a condizioni specificate dal programmatore che possono dipendere dai valori delle macro definite.

La principale direttiva usata per la compilazione condizionale è

`#if` *`expr`*

dove *`expr`* è un'espressione costante (non contenente variabili) intera. Tale espressione può ad esempio contenere costanti numeriche, macro, operatori logici, ecc., e in particolare può contenere uno speciale operatore messo a disposizione dal preprocessore,

`defined(`*`identificatore`*`)`

[1] Molti compilatori sono in grado di eseguire, quando lo ritengono opportuno, un'ottimizzazione che elimina il costo di una chiamata di funzione, copiando l'implementazione della funzione direttamente nel codice compilato del chiamante, e facendo ciò in un modo che non alteri il significato del programma. Questo permette in molti casi di ottenere il principale vantaggio delle macro con parametri senza tutti i loro svantaggi.

che restituisce il valore 1 se è definita una macro di nome *identificatore*, mentre restituisce 0 se invece non esiste una macro con tale nome. Con una direttiva `#if` inizia un blocco di codice che deve essere chiuso da una direttiva `#endif`:

```
#if expr
...
#endif
```

Questo blocco di codice viene:

- lasciato "così com'è" se la condizione della direttiva `#if` è vera (la valutazione di *expr* produce un valore diverso da 0);
- eliminato dall'input del compilatore, cioè non compilato, se la condizione dell'`#if` è falsa (la valutazione di *expr* produce il valore 0).

Tra `#if` e `#endif` si possono inoltre usare le direttive

```
#elif expr
#else
```

che individuano blocchi di codice alternativi da compilare se la condizione dell'`#if` è falsa; queste direttive hanno una semantica analoga a quella delle istruzioni di controllo `else if` ed `else` del linguaggio C (in particolare, la parola riservata `elif` è una contrazione di `else if`).

Infine, esistono delle forme abbreviate della direttiva `#if`, per i casi comuni in cui la condizione è il fatto che una singola macro sia definita o meno:

- `#ifdef` *identificatore* equivale a `#if defined(`*identificatore*`)`;
- `#ifndef` *identificatore* equivale a `#if !defined(`*identificatore*`)`.

3.1 Esempi

Se un frammento di codice deve essere compilato solo per le piattaforme identificate dalle macro `HP9000` e `SUN` (ciascuna delle quali è definita solo quando si sta producendo un eseguibile per tale piattaforma), si scrive:

```
#if defined(HP9000) || defined(SUN)
// Codice che dipende dalla macchina...
#endif
```

Se invece si hanno due frammenti di codice tra cui scegliere, uno per la piattaforma `SUN` e uno per tutte le altre piattaforme, si potrebbe ad esempio scrivere

```
#ifdef SUN
// Codice per la piattaforma SUN...
#else
// Codice per le altre piattaforme...
#endif
```

oppure, equivalentemente,

```
#if defined(SUN)
// Codice per la piattaforma SUN...
#else
// Codice per le altre piattaforme...
#endif
```

Considerando invece lo scenario delle istruzioni di debugging, se esse vengono racchiuse ad esempio con delle direttive `#if DEBUG`,

```
#if DEBUG
printf("Valore di a: %d\n", a);
#endif
```

allora vengono compilate se è presente la definizione

```
#define DEBUG 1
```

(che deve essere messa prima delle occorrenze di `#if DEBUG`, ad esempio in un file di intestazione), mentre non vengono compilate se è presente la definizione

```
#define DEBUG 0
```

o se la macro `DEBUG` non è proprio definita.

4 Direttiva `#undef`

La direttiva `#undef` permette di eliminare la definizione di una macro, "liberando" il corrispondente identificatore in modo che esso possa ad esempio essere usato per altre macro. In particolare, scrivere

`#undef` *identificatore*

ha l'effetto di eliminare la macro chiamata *identificatore*, se esiste, rendendola non più disponibile da questo punto del codice in poi. Se invece tale macro non esiste, la direttiva non ha alcun effetto (in particolare, non genera un errore).

Siccome l'inclusione di un file di intestazione comprende anche le definizioni di macro che esso contiene, eliminare una macro può essere utile, ad esempio, per evitare conflitti di nomi quando un file di intestazione incluso definisce una macro con lo stesso nome di una che vuole invece definire il programmatore:

```
#include "tutto.h" // Potrebbe definire una macro DIM
#undef DIM
#define DIM 100
```

5 Funzioni con un numero variabile di parametri

Il linguaggio C permette la definizione di funzioni che ricevono un numero variabile di parametri, come ad esempio `printf` e `scanf`. All'interno di tali funzioni, l'accesso ai parametri attuali avviene sostanzialmente a basso livello, accedendo direttamente al record di attivazione. Siccome il modo in cui i record di attivazione sono concretamente organizzati è lasciato a discrezione del compilatore, per consentire la scrittura di codice portabile lo standard C mette a disposizione alcune macro che permettono al programmatore di accedere ai parametri attuali senza conoscere tali dettagli (e ogni compilatore fornisce un'implementazione di queste macro che tiene conto dello specifico layout dei record di attivazione).

5.1 Elenco dei parametri formali

Nell'elenco dei parametri formali di una funzione, per indicare un numero variabile di parametri si usano tre punti, `...` (chiamati *ellipsis* in inglese). Si osservi che questa sintassi non specifica né i tipi né i nomi dei parametri che possono essere forniti in numero variabile: l'unico modo di ottenere i valori dei parametri attuali sono le macro di accesso al record di attivazione, e tali macro richiedono al programmatore di specificare il tipo di ciascun valore da recuperare, quindi serve una qualche convenzione per stabilire quali siano i tipi (e il numero) dei parametri in una determinata invocazione della funzione (ad esempio, `printf` conosce i tipi dei parametri attuali grazie alle specifiche di conversione presenti nella stringa di controllo).

Una funzione con un numero variabile di parametri *deve* avere anche almeno un parametro "normale", dotato di nome e tipo, che sarà sempre presente nelle chiamate (mentre i parametri in numero variabile possono essere anche zero). I parametri in numero variabile devono essere specificati dopo quelli sempre presenti, cioè i tre punti devono essere in ultima posizione nell'elenco dei parametri formali: ad esempio, il prototipo

```
int f(int a, float b, ...);
```

è corretto, mentre i prototipi

```
int f(int a, ..., float b);
int g(...);
```

non sono ammessi.

5.2 Macro di accesso

Il file di intestazione `stdarg.h` fornisce le macro per l'accesso alla lista dei parametri attuali. Queste macro vengono usate insieme a una variabile del tipo `va_list` (definito nello stesso file di intestazione), che punta di volta in volta a un argomento.

A scopo illustrativo, si supponga di voler implementare una funzione con il seguente prototipo:

```
float p(int a[], int n, ...);
```

Nel corpo di questa funzione, per accedere alla lista dei parametri attuali di numero variabile bisogna innanzitutto definire una variabile di tipo `va_list` (con un nome di propria scelta):

```
va_list ap;
```

Si usa poi la macro `va_start` per inizializzare questa variabile, in modo che punti al primo parametro "senza nome". Tale macro riceve due argomenti: la variabile da inizializzare (che qui è `ap`) e l'ultimo parametro dotato di nome (che qui è `n`):

```
va_start(ap, n);
```

siccome i parametri sono memorizzati uno dopo l'altro nel record di attivazione, sapendo l'indirizzo e il tipo dell'ultimo parametro dotato di nome è possibile ricavare l'indirizzo del primo parametro senza nome (ad esempio, tale indirizzo potrebbe essere `&n` + `sizeof(n)` byte).

Adesso, è possibile utilizzare la macro `va_arg` per accedere in sequenza a ciascun parametro attuale. Essa riceve come argomenti la variabile `ap` e il tipo dell'argomento da leggere, ed esegue (almeno concettualmente) le seguenti operazioni:

1. legge il parametro a cui punta `ap`, interpretando il valore presente in memoria come un valore del tipo specificato;
2. incrementa `ap` in modo che punti al parametro successivo (facendo avanzare il puntatore di un numero di byte che dipende dal tipo specificato);
3. restituisce il valore letto.

Ad esempio:

```
int i = va_arg(ap, int);
```

Infine, prima che la funzione restituisca il controllo al chiamante è necessario usare la macro

```
va_end(ap);
```

che a seconda dell'implementazione potrebbe servire, ad esempio, a ripristinare lo stato del record di attivazione in modo da poter effettuare correttamente l'operazione di `return`.

5.3 Esempio: somma di un numero variabile di valori

Il seguente programma illustra la definizione e l'uso di una funzione `sum` che esegue la somma di un numero arbitrario di valori `int` e `double`:

```
#include <stdarg.h>
#include <stdio.h>

double sum(const char *types, ...) {
    va_list ap;
    va_start(ap, types);
    double sum = 0.0;
    for (int i = 0; types[i] != '\0'; i++) {
        if (types[i] == 'i') sum += va_arg(ap, int);
        else if (types[i] == 'd') sum += va_arg(ap, double);
        else break;
    }
    va_end(ap);
    return sum;
}

int main(int argc, const char *argv[]) {
    double a = 73.25;
    int b = 100;
    printf("%f\n", sum("diiidii", 100.78, 3, 5, 7, a, 9, b));
    return 0;
}
```

Come detto prima, quando si definisce una funzione con un numero variabile di parametri è necessario stabilire una qualche convenzione per indicare alla funzione il numero e i tipi dei parametri forniti. In questo caso, si è scelto di usare a tale scopo una stringa, cioè un vettore di caratteri terminato dal carattere nullo `'\0'`, il cui indirizzo viene passato come primo parametro a `sum`. Ciascun carattere di questa stringa indica il tipo di un parametro senza nome: `'i'` significa `int` e `'d'` significa `double`.

Per eseguire il calcolo, la funzione `sum` si occupa per prima cosa di definire e inizializzare la variabile `va_list ap`, dopodiché scorre i caratteri della stringa passata come primo parametro, leggendo per ciascun carattere un parametro senza nome del tipo specificato, che viene aggiunto alla variabile di somma. Se si incontra un carattere non

previsto, diverso da `'i'` o `'d'`, la scansione termina immediatamente, senza raggiungere il terminatore della stringa (`'\0'`).

Infine, `sum` restituisce la somma calcolata al chiamante, dopo aver usato `va_end` per assicurarsi che sia possibile eseguire correttamente l'operazione di `return`.

All'interno di `main`, la funzione `sum` viene chiamata in questo modo:

```
sum("diiidii", 100.78, 3, 5, 7, a, 9, b)
```

La stringa `diiidii` specifica che sono forniti 7 parametri senza nome: un valore `double`, tre `int`, ancora un `double`, e altri due `int`. Siccome qui il numero e i tipi dei parametri indicati nella stringa corrispondono al numero e ai tipi dei parametri effettivamente passati, funziona tutto correttamente: il programma stampa il risultato `298.030000`.

La somma funzionerebbe correttamente anche se i parametri specificati nella stringa fossero meno di quelli passati: semplicemente, i parametri in più verrebbero ignorati. Ad esempio, la chiamata

```
sum("diiid", 100.78, 3, 5, 7, a, 9, b)
```

restituisce correttamente il risultato `189.030000`, ignorando gli ultimi due parametri.

Se invece i parametri specificati nella stringa fossero più di quelli passati, e/o se i tipi non corrispondessero, il compilatore non avrebbe modo di accorgersene, e in esecuzione si avrebbero risultati imprevedibili, perché i byte dei parametri attuali verrebbero interpretati in modo sbagliato e/o il programma tenterebbe di leggere byte successivi a quelli dell'ultimo parametro. Ad esempio, la chiamata

```
sum("diiidii", 100, 3, 5, 7, a, 9, b)
```

non è corretta perché il primo parametro senza nome è una costante di tipo `int`, mentre la stringa specifica un parametro di tipo `double`.

5.4 Esempio: `printf` minimale

Un altro esempio di funzione (procedura) con un numero variabile di parametri è un'implementazione minimale di `printf`, che riconosce solo le specifiche di conversione `%d`, `%f` e `%s`:

```
#include <stdarg.h>
#include <stdio.h>

void minprintf(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    for (const char *p = fmt; *p; p++) {
        if (*p != '%') {
```

```
            putchar(*p);
            continue;
        }
        p++;
        switch (*p) {
            case 'd':
                printf("%d", va_arg(ap, int));
                break;
            case 'f':
                printf("%f", va_arg(ap, double));
                break;
            case 's':
                for (const char *s = va_arg(ap, const char*); *s; s++)
                    putchar(*s);
                break;
            default:
                putchar('%');
                putchar(*p);
                break;
        }
    }
    va_end(ap);
}

int main(int argc, const char *argv[]) {
    int a = 8;
    double b = 5.75;
    minprintf("a vale %d e b vale %f\n", a, b);
    minprintf("Stampa di una stringa: %s\n", "xyz");
    return 0;
}
```

Come nell'esempio precedente, questa procedura scorre la stringa di controllo passata come primo parametro per determinare il numero e i tipi dei parametri senza nome da leggere. La scansione avviene tramite il puntatore `p`, che a ogni iterazione del ciclo `for` punta a un carattere successivo della stringa; quando `p` arriva a puntare al carattere terminatore della stringa, cioè quando `*p == '\0'`, la condizione `*p` risulta falsa, dunque il ciclo termina.

Innanzitutto, ogni carattere della stringa di controllo che non fa parte di una specifica di conversione viene riportato direttamente in output, usando la funzione `putchar` per stamparlo. Se invece si incontra il carattere `%`, si fa avanzare il puntatore `p` per leggere la lettera che segue, e in base a essa si determina che tipo di parametro leggere e stampare. Per semplicità, si evita di implementare la conversione dei valori `int` e `double` in stringhe, richiamando semplicemente la funzione di libreria `printf` per stamparle (ma ovviamente

questo non è possibile nella vera implementazione di `printf`). Invece, i parametri di tipo stringa vengono stampati un carattere alla volta, usando `putchar` in modo simile a quanto fatto per la stringa di controllo. Infine, eventuali specifiche di conversione non riconosciute vengono semplicemente riportate in output.