Azzolini Riccardo 2021-03-01

Istruzioni di controllo e funzioni di I/O

1 Istruzioni di controllo

Il linguaggio C prevede diverse istruzioni per regolare il flusso del controllo, che sono in gran parte analoghe a quelle presenti, ad esempio, in Java.

1.1 If-then

L'istruzione **if-then** è il selettore a una via, che esegue un'istruzione solo se è verificata una certa condizione. In C, essa ha la sintassi

`if (`*`expr`*`)` *`statement`*

dove *`expr`* è una qualunque espressione C e *`statement`* è un'arbitraria istruzione, che può essere semplice o composta (ovvero un blocco di istruzioni racchiuso tra parentesi graffe). Si noti in particolare che, per brevità, la sintassi non prevede la parola riservata `then` (presente in molti altri linguaggi esistenti al tempo della progettazione del C): lo *`statement`* inizia implicitamente dopo la parentesi chiusa che segue *`expr`*.

La *semantica* (operazionale) di quest'istruzione, cioè il suo significato in termini di operazioni eseguite, è la seguente: viene valutata l'espressione *`expr`*, e

- se il risultato è un valore diverso da 0, che rappresenta il valore di verità *vero*, viene eseguita l'istruzione *`statement`*;
- se invece il risultato è 0, che rappresenta il valore di verità *falso*, allora *`statement`* non viene eseguita, e si passa direttamente all'esecuzione della prima istruzione che segue il costrutto if-then.

Un esempio di istruzione if-then è

```
if (a < 0) printf("Attenzione: a negativo!\n");
```

la cui esecuzione causa la stampa di un messaggio se la variabile `a` ha un valore negativo.

1.2 If-then-else

L'istruzione **if-then-else** è il selettore a due vie, che si presenta con la sintassi

`if (`*`expr`*`)` *`statement1`* `else` *`statement2`*

dove *`expr`* è un'espressione, mentre *`statement1`* e *`statement2`* sono arbitrarie istruzioni (semplici o composte).

Semantica: Viene valutata *`expr`*, ed eseguita l'istruzione

- *`statement1`* se il risultato della valutazione è un valore diverso da 0 (*vero*);
- *`statement2`* se il risultato è 0 (*falso*).

Un esempio è

```
if (a < b) c = b; else c = a;
```

che assegna alla variabile `c` il massimo tra i valori di `a` e `b`.

1.2.1 Operatore `?:`

L'operatore ternario `?:` (analogo a quello presente, ad esempio, in Java) è un'alternativa compatta all'istruzione if-then-else, che permette di costruire *espressioni condizionali* il cui valore dipende da una condizione. La sua sintassi è

`expr1` `?` *`expr2`* `:` *`expr3`*

dove *`expr1`*, *`expr2`* e *`expr3`* sono arbitrarie espressioni.

Semantica: Viene valutata l'espressione *`expr1`*, e l'espressione condizionale assume il valore ottenuto dalla valutazione dell'espressione

- *`expr2`* se il valore di *`expr1`* è diverso da 0 (cioè vero);
- *`expr3`* se il valore di *`expr1`* è 0 (falso).

Ad esempio, l'istruzione

```
i = a < b ? a : b;
```

assegna alla variabile `i` il minimo tra i due valori `a` e `b`, ed equivale a:

```
if (a < b) i = a; else i = b;
```

1.3 Switch

L'istruzione **switch** è un selettore a più vie, la cui sintassi è

```
switch (expr) {
    case const1: statement1 break;
    ...
    case constN: statementN break;
    default: statement
}
```

dove *expr* è un'espressione avente valore intero, *const1*, ..., *constN* sono valori costanti interi e *statement1*, ..., *statementN*, *statement* sono istruzioni arbitrarie. Il caso `default` è opzionale.

Semantica: Viene valutata l'espressione *expr*, ed eseguita l'istruzione corrispondente al valore costante uguale al valore di *expr*, se esiste. Quando invece nessuna delle costanti specificate è uguale al valore di *expr*, viene eseguita l'istruzione *statement* associata al caso `default`, se presente.

L'istruzione `break;` che termina ciascun caso verrà spiegata più in dettaglio a breve, ma in sostanza, in questo caso, serve a far saltare l'esecuzione all'istruzione successiva al costrutto switch, invece di proseguire con l'istruzione del caso successivo.

Un esempio di switch è

```
switch (ch) {
    case 'a': cont_a++; break;
    case 'e': cont_e++; break;
    case 'i': cont_i++; break;
    case 'o': cont_o++; break;
    case 'u': cont_u++; break;
    default: cont_car++;
}
```

che seleziona il contatore da incrementare in funzione del valore della variabile `ch`. Si noti che è ammesso specificare caratteri come costanti per i vari casi poiché, come già detto, il tipo carattere è a tutti gli effetti un tipo intero, ovvero un carattere viene interpretato come il valori numerico della sua codifica.[1]

[1] Inoltre, per motivi storici, i caratteri scritti tra singoli apici vengono interpretati come valori di tipo `int`, anziché `char`.

1.4 While

L'istruzione **while** ha la sintassi

```
while (expr) statement
```

dove *expr* è una qualunque espressione e *statement* è un'arbitraria istruzione.

Semantica: Per prima cosa, viene valutata *expr*. Se il risultato della valutazione è un valore diverso da 0, si effettua la prima iterazione del ciclo, cioè viene eseguita l'istruzione *statement*, dopodiché si torna a valutare *expr* per determinare se eseguire una seconda iterazione, e così via. Le iterazioni terminano, e l'esecuzione passa alla prima istruzione dopo il while, quando la valutazione di *expr* dà risultato 0.

Perché un ciclo while non sia infinito, è necessario che l'istruzione *statement* modifichi il valore di qualche variabile che compare nell'espressione *expr* (o che *statement* contenga un'istruzione `break`, come si vedrà a breve).

Un esempio di ciclo while è il seguente frammento di codice C,

```
while (i < n && A[i] < 1000) i = i + 1;
```

che calcola l'indice del primo valore maggiore o uguale a 1000 presente in un vettore `A` di `n` elementi.

1.5 Do-while

L'istruzione **do-while** è un ciclo nel quale la condizione viene verificata alla fine di (*in coda* a) ciascuna iterazione, invece che all'inizio (*in testa*), come nel caso del ciclo while. La sua sintassi è

```
do statement while (expr);
```

dove *expr* è una qualunque espressione e *statement* è una qualunque istruzione.

Semantica: Per prima cosa, si esegue l'istruzione *statement*, dopodiché viene valutata l'espressione *expr*, e se il risultato è diverso da 0 si prosegue con una nuova iterazione, eseguendo di nuovo *statement* e poi valutando di nuovo *expr*. Il ciclo termina, passando il controllo alla prima istruzione successiva al costrutto do-while, quando *expr* assume valore 0.

Sostanzialmente, la differenza tra while e do-while è che quest'ultimo garantisce l'esecuzione di almeno un'iterazione, mentre while non esegue alcuna iterazione se la condizione è falsa (*expr* vale 0) fin dall'inizio.

Ad esempio, un ciclo do-while può essere usato per richiedere all'utente di inserire un valore intero pari:

```c
do {
    printf("Immettere un valore intero pari\n");
    scanf("%d", &a);
} while (a % 2);
```

- La funzione `scanf` verrà ripresa tra poco, ma in questo caso essa richiede un input dallo standard input (che di solito è la tastiera), lo converte in un valore numerico intero e lo memorizza nella variabile `a`.
- L'espressione `a % 2`, usata come condizione del ciclo, calcola il resto ottenuto dividendo `a` per 2: se `a` è pari, allora tale resto è 0 (per definizione), quindi il ciclo termina, mentre se `a` è dispari il resto è diverso da 0, dunque il ciclo continua, chiedendo all'utente di inserire un nuovo valore.

1.6 For

L'istruzione **for** è un ciclo che ha la sintassi

`for (`*`expr1`*`; `*`expr2`*`; `*`expr3`*`)` *`statement`*

dove *`expr1`*, *`expr2`* e *`expr3`* sono espressioni, e *`statement`* è una qualunque istruzione.

Semantica: Per iniziare viene eseguita l'espressione *`expr1`*, che rappresenta l'azione di inizializzazione del ciclo, ed è appunto eseguita una sola volta. Poi, all'inizio di ciascuna iterazione (compresa la prima), viene valutata *`expr2`*, che rappresenta la condizione del ciclo: se essa ha valore diverso da 0, allora viene eseguita l'istruzione *`statement`*, dopodiché viene valutata l'espressione *`expr3`*, che tipicamente serve ad aggiornare una variabile che compare nella condizione *`expr2`*. Infine, si torna a valutare *`expr2`* per determinare se eseguire una nuova iterazione, e il processo si ripete fino a quando *`expr2`* assume valore 0.

In sintesi, un ciclo for equivale a una forma particolare di ciclo while:

`expr1``; while (`*`expr2`*`) { `*`statement`*`; `*`expr3`*`; }`

Dal punto di vista stilistico, si preferisce solitamente usare:

- un ciclo for quando il numero di iterazioni da eseguire è noto a priori;
- un ciclo while quando invece il numero di iterazioni è variabile.

Un esempio di ciclo for è il seguente, che stampa i valori di $\mathtt{a}^{2^{\mathtt{i}}}$ per $\mathtt{i} = 1, \ldots, \mathtt{n} - 1$:

```c
for (i = 1; i < n; i++) {
    a = a * a;
    printf("a^(2^%d) = %d", i, a);
}
```

Il ciclo while equivalente è:

```
i = 1;
while (i < n) {
    a = a * a;
    printf("a^(2^%d) = %d", i, a);
    i++;
}
```

1.7 Break

L'istruzione **break** viene utilizzata per alterare il normale flusso esecutivo all'interno di un blocco di istruzioni. Essa può comparire solo nel corpo di uno switch, while, do-while o for (eventualmente all'interno di un blocco di istruzioni annidato nel corpo di uno di questi costrutti):

```
switch (expr) { ... break; ... }
 while (expr) { ... break; ... }
           do { ... break; ... } while (expr);
for (expr1; expr2; expr3) { ... break; ... }
```

Semantica: L'esecuzione di un'istruzione break causa un salto (trasferimento del controllo) incondizionato alla prima istruzione che segue il costrutto switch, while, do-while o for che contiene quest'istruzione.[2]

Ad esempio, il seguente frammento di codice C stampa il numero di caratteri letti dallo standard input prima di incontrare il carattere `'n'`, fermandosi però se non trova un carattere `'n'` preceduto da al massimo 100 caratteri:

```
i = 0;
while (1) {
    c = getchar();
    if (c == 'n') {
        printf("Carattere n preceduto da %d caratteri\n", i);
        break;
    }
    i = i + 1;
    if (i > 100) break;
}
```

[2] Se l'istruzione break è contenuta in uno switch o ciclo annidato dentro un altro switch o ciclo, viene considerato il costrutto più interno.

Si noti che questo è un ciclo apparentemente infinito (l'espressione `1` usata come condizione di iterazione ha sempre valore vero), nel quale le "vere" condizioni di uscita sono specificate tramite i break presenti nel corpo. Nella pratica si usano spesso cicli fatti in questo modo, poiché consentono di esprimere condizioni di uscita complesse in modo più chiaro rispetto all'uso di una singola espressione complessa come condizione del while.

1.8 Continue

L'istruzione **continue** altera il normale flusso esecutivo all'interno di un ciclo (while, do-while o for):

```
while (expr) { ... continue; ... }
do { ... continue; ... } while (expr);
for (expr1; expr2; expr3) { ... continue; ... }
```

Semantica: Quando viene eseguita, un'istruzione continue causa un salto (trasferimento del controllo) incondizionato alla valutazione dell'espressione che condiziona l'esecuzione del ciclo in cui tale istruzione si trova, cioè di fatto provoca l'interruzione dell'iterazione corrente e l'inizio dell'iterazione successiva (se il risultato della valutazione della condizione è diverso da 0).

Un esempio di uso di continue è il seguente frammento di codice, che stampa la frequenza del carattere `'e'` in una sequenza di lunghezza `NUMCHAR` di caratteri letti dallo standard input:

```
cont = i = 0;
while (i <= NUMCHAR) {
    i = i + 1;
    c = getchar();
    if (c != 'e') continue;
    cont++;
}
freq = ((float)cont) / NUMCHAR;
printf("Frequenza del carattere e: %f\n", freq);
```

1.9 Goto

L'istruzione **goto** altera il normale flusso esecutivo, indicando un salto incondizionato all'istruzione associata all'etichetta specificata. Essa ha la sintassi

```
...
goto label;
...
label: statement
```

dove *label* è un'etichetta (un arbitrario identificatore) e *statement* è una qualunque istruzione.

Semantica: L'esecuzione di un'istruzione goto causa il trasferimento del controllo all'istruzione associata all'etichetta specificata nel goto.

Il goto va utilizzato con parsimonia, altrimenti porta molto facilmente alla scrittura di codice illeggibile, poiché va contro i principi della programmazione strutturata. Tuttavia, ci sono alcune situazioni (abbastanza rare) in cui usare il goto rende invece il codice più compatto e leggibile.

Utilizzando le sole istruzioni goto e if-then, è possibile realizzare tutte le altre istruzioni di controllo (e questo è proprio ciò che fa il compilatore quando traduce un programma C in linguaggio macchina, che ha solo i salti incondizionati e condizionati come istruzioni di controllo). Ad esempio, il ciclo while

```
i = 0;
while (i <= NUM)
    printf("%d\n", ++i);
// ...
```

ammette la seguente traduzione tramite l'istruzione goto:

```
        i = 0;
inizio: if (i > NUM) goto fine;
        printf("%d\n", ++i);
        goto inizio;
fine:   // ...
```

2 Input e output

Il C dispone di numerose funzioni di libreria per effettuare operazioni di input e output. In seguito, verranno presentate alcune delle funzioni di base.

2.1 `printf`

La funzione `printf` permette di eseguire l'**output formattato** (la "f" sta appunto per "formatted") di un elenco di dati di lunghezza variabile sullo **standard output** (che di solito corrisponde al monitor). Il suo primo parametro, che è obbligatorio, è una **stringa di controllo** che specifica il numero e i tipi dei dati da stampare e il modo in cui formattarli, mentre i parametri successivi costituiscono l'elenco di tali dati, che può avere una lunghezza qualsiasi (anche 0, cioè si può specificare anche solo la stringa di controllo, se si vuole ottenere un output predeterminato senza dati formattati).

La stringa di controllo è una sequenza di caratteri (indicata tra doppi apici, con la solita sintassi delle stringhe) che può contenere una qualunque combinazione di:

- dei caratteri "normali", da trasferire direttamente in output;
- delle **specifiche di conversione**, in pratica dei segnaposto (placeholter) che nell'output verranno sostituiti dai valori delle variabili/espressioni passate a `printf`.

Una specifica di conversione è una stringa che inizia con il simbolo `%`, e termina con un *carattere di conversione* che descrive il tipo del valore da stampare e la notazione con cui stamparlo. Alcuni dei principali caratteri di conversione sono:

Carattere di conversione	Significato
`c`	carattere
`d`	numero intero in notazione decimale
`f`	numero reale in notazione decimale
`s`	stringa
`p`	puntatore in notazione esadecimale

Le specifiche di conversione presenti nella stringa di controllo devono essere lo *stesso numero*, specificare gli *stessi tipi* ed essere nello *stesso ordine* delle variabili/espressioni passate come ulteriori parametri a `printf`, altrimenti:

- Se si usasse una specifica di conversione di tipo diverso da quello del corrispondente parametro, allora i byte del valore del parametro verrebbero interpretati in modo sbagliato. Ad esempio, data una variabile `x` di tipo `double`, e supponendo che le dimensioni di `double` e `int` siano rispettivamente 8 e 4 byte, l'istruzione

  ```
  printf("%d", x);
  ```

  interpreta come numero intero i primi 4 byte del valore di `x`, dunque stampa un valore apparentemente "senza senso".

- Se ci fossero più specifiche di conversione che parametri, verrebbero probabilmente stampati dei valori letti da qualche locazione di memoria (probabilmente situati nel record di attivazione di `printf`), oppure si potrebbe verificare un errore in esecuzione (se ad esempio tale locazione di memoria fosse protetta dal sistema operativo);
- Se invece ci fossero più parametri che specifiche di conversione, i parametri in eccesso verrebbero semplicemente ignorati, senza causare problemi.

Un esempio di uso di `printf` è il seguente:

```
printf("Il carattere %c occorre %d volte\n", ch, cont);
```

Esso stampa sullo standard output, in ordine:

1. la stringa `"Il carattere "`,
2. il carattere corrispondente al valore della variabile `ch`,
3. la stringa `" occorre "`,
4. la rappresentazione decimale del valore della variabile intera `cont`,
5. la stringa `" volte"`,
6. un newline (a capo).

2.2 `scanf`

La funzione `scanf` effettua l'**input formattato** dallo **standard input** (che di solito è la tastiera), permettendo di leggere un elenco di dati di lunghezza variabile. Similmente a `printf`, essa ha un parametro obbligatorio, la stringa di controllo, seguito da una sequenza di zero o più variabili nelle quali verranno salvati i dati letti.

La stringa di controllo di `scanf`, che ha un formato analogo a quello di `printf`, serve a specificare il numero di dati da leggere e le conversioni da effettuare per la lettura, che devono coincidere con il numero e i tipi delle variabili specificate per il salvataggio dei dati.

Una volta letti i dati, per poterli salvare la funzione `scanf` ha bisogno di conoscere gli *indirizzi*, e non i valori, delle variabili in cui salvarli. Di conseguenza, se ad esempio si vuole salvare un numero intero in una variabile `n`, bisogna passare come argomento non direttamente `n` (questo sarebbe il valore della variabile), ma piuttosto `&n`:

```
scanf("%d", &n);
```

`&` è l'**operatore di estrazione di indirizzo**, che restituisce appunto l'indirizzo della variabile a cui è applicato. Se si dimentica `&` nell'invocazione di `scanf`,

```
scanf("%d", n);
```

allora il valore della variabile `n` viene interpretato come un indirizzo di memoria in cui scrivere dei dati, il che comporta conseguenze simili a quelle dell'accesso a un indice non valido di un array. Ad esempio:

- potrebbe verificarsi un errore in esecuzione, ad esempio se questo indirizzo rientra in una zona di memoria protetta dal sistema operativo;
- potrebbe essere modificato, in modo imprevedibile e senza alcuna avvertenza, il valore di una qualche altra variabile del programma.

Se la stringa di controllo contiene caratteri "normali" (che non rappresentano specifiche di conversione), `scanf` si aspetta che l'utente inserisca in input letteralmente quei caratteri, altrimenti fallisce e non legge alcun dato. Ad esempio, si consideri la seguente istruzione:

```
scanf("Intero: %d", &n);
```

- Se l'utente inserisce, ad esempio,

  ```
  Intero: 37
  ```

  allora `scanf` assegna il valore 37 alla variabile `n`.
- Se invece l'utente inserisce solo `37`, `scanf` fallisce e non assegna nulla a `n`, perché si aspettava di leggere un input che iniziasse con il carattere `'I'`, e non con una cifra decimale.

2.3 Input e output non formattato

Le funzioni di input e output non formattato permettono di effettuare letture e scritture "carattere per carattere". Le principali sono `getc` e `putc`.

- `getc` legge un singolo carattere (`unsigned char`) dallo **stream** di input specificato come argomento. Uno stream è una successione di byte (caratteri), che può ad esempio essere un file o lo standard input. In particolare, per indicare lo standard input si passa come argomento `stdin`: `getc(stdin)`.
- `putc` scrive un singolo carattere (`unsigned char`), passato come primo argomento, sullo stream indicato come secondo argomento. Qui si può usare `stdout` per indicare lo standard output: ad esempio, l'istruzione

  ```
  putc('Z', stdout);
  ```

  stampa il carattere `'Z'` sullo standard output.

Per comodità, `getc` e `putc` hanno delle versioni specializzate per gli stream di standard input e output, chiamate `getchar` e `putchar`:

- `getchar()` equivale a `getc(stdin)`;

- `putchar(c)`, dove `c` è una variabile o espressione di tipo carattere, equivale a `putc(c, stdout)`.