Informatica per Statistica
Riassunto della lezione del 02/11/2012

Igor Melatti

Altre istruzioni e costrutti del C

- Due nuove tipologie di istruzioni: il `do..while` e lo `switch..case`
  - sintassi del `do..while`:

    `do <blocco_istruzioni> while (<condizione>);`
  - **l'effetto è quello di eseguire <blocco_istruzioni>, dopodiché si controlla se <condizione> è vera; in caso positivo, si esegue nuovamente <blocco_istruzioni> per poi controllare nuovamente se <condizione> è vera, e così via (<blocco_istruzioni>, poi <condizione>, poi <blocco_istruzioni>, ...); non appena uno di questi controlli su <condizione> risulta falsa, l'istruzione `do...while` termina e si passa alla successiva**
  - è quindi come il `while`, tranne per il fatto che <blocco_istruzioni> viene sicuramente eseguito almeno una volta (la prima)
  - nel `while`, <blocco_istruzioni> potrebbe anche non essere eseguito mai
  - la sintassi dello `switch..case` è data in Figura 1

```
1 switch (<espressione_intera_var>) {
2   case <espressione_intera_cost>:
3     <blocco_istruzioni>
4   ... /* un certo numero di casi */
5   case <espressione_intera_cost>:
6     <blocco_istruzioni>
7   default:
8     <blocco_istruzioni>
9 }
```

Figure 1: Sintassi dello `switch..case`

    - * <`espressione_intera_var`> può contenere variabili, ma di tipo intero o carattere
    - * <`espressione_intera_cost`> non può contenere variabili
    - * è possibile ripetere il costrutto `case` <`espressione_intera_cost`>`:` <`blocco_istruzioni`> tutte le volte che si vuole
    - * ma sempre con valori diversi per <`espressione_intera_cost`>
    - * il caso `default` può non essere presente, e non necessariamente è l'ultimo caso
  - per la semantica, si supponga di avere un'istruzione `switch` (E) $\{$`case` v_1`:` I_1`;` ... `case` v_n`:` I_n`; default:` $I\}$
  - **l'effetto è quello di valutare E; se esiste un $i \in [1, n]$ t.c. E == v_i, allora si eseguono in sequenza $I_i, I_{i+1}, \ldots I_n, I$; altrimenti (ovvero, se nessuno dei v_i è uguale ad E) si esegue I**
    - * quindi se l'espressione data nello `switch` è uguale ad un qualche valore dato nei `case`, si eseguono tutte le istruzioni a partire da quel punto
    - * se nessun valore dato nei `case` è quello giusto, allora si esegue il `default` (se c'è), e le eventuali istruzioni seguenti (non è detto che il default sia l'ultimo caso)
  - questa semantica non corrisponde con il senso "intuitivo" di uno `switch`..`case`: ci si aspetterebbe che se E è uguale ad un v_i, allora si esegue solo I_i
  - per far sì che questo succeda, basta usare propriamente l'istruzione `break`

- **L'istruzione `break` ha l'effetto di interrompere il ciclo o lo switch più interno nel quale venga eseguita; dopodiché si continua con l'istruzione successiva al ciclo o switch interrotto**
  - è pertanto sufficiente posizionare un `break` alla fine del blocco istruzioni per far sì che venga eseguito solo quello

- Esempio di `switch` senza `break`: Figura 2
  - sintatticamente è corretto, ma non risolve il problema di scrivere a lettere i numeri fino a 3
  - infatti, se per esempio `n` è 2, allora stamperà
    ```
    2 e' due
    2 e' tre
    2 e' e' piu' grande di tre
    ```
  - questo perché, considerando la semantica dello `switch`, nell'esecuzione il programma trova che il `case 2` è quello che soddisfa l'uguaglianza con `n`, dopodiché esegue tutte le istruzioni che seguono (ignorando gli altri `case`)

```
1  void numero_a_parole(unsigned n)
2  {
3    switch (n) {
4      case 1:
5        printf("%u e' uno", n);
6      case 2:
7        printf("%u e' due", n);
8      case 3:
9        printf("%u e' tre", n);
10     default:
11       printf("%u e' piu' grande di tre", n);
12   }
13 }
```

Figure 2: Esempio di `switch` (sintatticamente giusto, semanticamente sbagliato)

```
1  void numero_a_parole(unsigned n)
2  {
3    switch (n) {
4      case 1:
5        printf("%u e' uno", n);
6        break;
7      case 2:
8        printf("%u e' due", n);
9        break;
10     case 3:
11       printf("%u e' tre", n);
12       break;
13     default:
14       printf("%u e' piu' grande di tre", n);
15       break; /* inutile */
16   }
17 }
```

Figure 3: Esempio di `switch` (sintatticamente e semanticamente giusto)

- Esempio di `switch` con `break`: Figura 2
  - è sia sintatticamente che semanticamente corretto (risolve il problema di scrivere a lettere i numeri fino a 3)
  - infatti, se per esempio `n` è 2, allora stamperà `2 e' due`
  - questo perché, considerando la semantica dello `switch`, nell'esecuzione il programma trova che il `case 2` è quello che soddisfa l'uguaglianza con `n`, dopodiché esegue tutte le istruzioni che seguono (ignorando gli altri `case`); ma dopo aver eseguito la prima `printf` trova un `break`, che lo forza ad uscire dallo `switch` senza eseguire nessun'altra istruzione
- Altre due istruzioni simili al `break`: `return` e `continue`
- **L'istruzione `return` <espressione> ha l'effetto di interrompere la funzione nel quale venga eseguita, e di ritornare la valutazione di <espressione>; dopodiché si continua con l'istruzione successiva alla corrispondente chiamata di funzione**

```
1  int f2(int arg1)
2  {
3    int c = arg1;
4
5    return c;
6  }
7
8  int f1(int arg1)
9  {
10   int a, b, c;
11
12   a = c + b*arg1;
13   return a;
14 }
```

Figure 4: Scoping di variabili (correzione 1)

  - se la chiamata di funzione si trovava all'interno di una espressione più complessa, allora si procede a valutare il resto di questa espressione
  - ad esempio, nella seguente chiamata a funzione: `var = 4*f1(var2) + var3;`, dopo che `f1` ha tornato il suo valore, occorre ancora moltiplicarlo per 4 e sommargli il valore di `var3`
  - <`espressione`> deve essere dello stesso tipo del tipo di ritorno della funzione dentro alla quale la `return` viene invocata

- ad esempio, le funzioni `f1` ed `f2` di Fig. 4 hanno come tipo di ritorno `int`, e infatti le `return` al loro interno hanno come espressione le variabili `c` ed `a`, che sono intere
- se il tipo della funzione in cui la `return` si trova è `void`, allora l'espressione non va messa
- la return non è obbligatoria, soprattutto per le funzioni che ritornano `void`: vedere Fig. 2

- **L'istruzione `continue` ha l'effetto di interrompere l'esecuzione del blocco di istruzioni del ciclo più interno nel quale venga eseguita, tornando subito alla condizione del ciclo stesso; nel caso si tratti di un ciclo `for`, viene prima effettuata l'istruzione di iterazione**
- **Esercizio**: con riferimento alla Figura 4, aggiungere delle inizializzazioni (a piacere) per le variabili non inizializzate, un main, e scrivere altre 2 funzioni `f1_bis` ed `f2_bis` che siano equivalenti a `f1` ed `f2`, ma che contengano una sola istruzione ciascuna

Costrutti con sintassi diversa e uguale semantica

- I tre tipi di cicli del C (ovvero il `while`, il `do..while` ed il `for`) hanno lo stesso potere espressivo
- Ovvero, è possible scrivere un qualsiasi ciclo scritto con una delle tre sintassi usando un ciclo scritto in un'altra sintassi
- È possibile prendere un ciclo `for` e scrivere un ciclo `while` con la stessa semantica (ovvero, che fa la stessa cosa)
  - un generico ciclo `for` sarà scritto così
    `for (`In`; `E`; `It`)` I
    - si intende che In indica una qualsiasi istruzione di inizializzazione, It una qualsiasi istruzione di iterazione, E una qualsiasi condizione booleana, e I un qualsiasi blocco di istruzioni
  - un ciclo `while` con la stessa semantica è il seguente:
    In`; while (`E`) { `I`; `It`; }`
  - infatti In viene eseguito una volta sola e per primo; ogni ciclo è condizionato al fatto che E sia vera; alla fine del blocco di istruzioni I, e prima di ricontrollare nuovamente E si esegue It
  - **esercizio**: considerare un programma contenente un ciclo `for`, riscriverlo con un ciclo `while` come sopra e controllare che funzioni
- È possibile prendere un ciclo `while` e scrivere un ciclo `do..while` con la stessa semantica (ovvero, che fa la stessa cosa)

- un generico ciclo `while` sarà scritto così
  `while (`E`)` I
- un ciclo `do..while` con la stessa semantica è il seguente:
  `do {if (`E`)` I`} while (`E`)`
- sostanzialmente, dato che l'unica differenza tra i due cicli consiste nel fatto che nel `do..while` almeno una volta I viene eseguito sempre, si fa in modo che ciò accada solo se E è vera, come nel `while`
- tuttavia, così facendo, per ogni ciclo si controlla E due volte: una dentro l'`if`, e una dentro il `while`
- E potrebbe contenere anche chiamate a funzioni complesse, quindi questa soluzione è assai più inefficiente del ciclo `while`
- se poi E contiene chiamate a funzioni che modificano variabili globali, allora potrebbe tranquillamente succedere che i due cicli, così scritti, non siano affatto equivalenti
- un modo per uscire da questa inefficienza (o scorrettezza), e controllare l'espressione E per lo stesso numero di volte per cui era controllata nel `while`, è illustrato in Figura 5

```
1 {
2   int prima_volta = 1; /* 1 vuol dire vero */
3   do {
4     if (prima_volta && E) {
5       prima_volta = 0; /* 0 vuol dire falso */
6       I
7     }
8   } while (E);
9 }
```

Figure 5: Ciclo `do..while` equivalente ad un ciclo `while`

- in Figura 5, si apre un nuovo blocco di istruzioni per poter dichiarare una variabile `prima_volta`
- pur essendo di tipo intero, concettualmente questa variabile viene usata come un booleano
- è infatti vera (cioè uguale ad 1) durante la prima iterazione del `do..while`, ed è falsa in tutte le altre (eventuali) iterazioni
- dopodiché, si sfrutta la *valutazione cortocircuitata del connettore logico AND*
  - dalla definizione dell'AND, risulta che 0 AND 0 = 0, e che 0 AND 1 = 0
  - dove, come nel C, si assume che 0 sia falso e 1 sia vero

        * quindi vuol dire che 0 AND $x = 0$, sia che $x = 0$ sia che $x = 1$
        * la valutazione cortocircuitata sfrutta questa proprietà di modo che, se occorre valutare un'espressione con un AND, e la prima parte è falsa, allora non si valuta affatto la seconda, ma si può direttamente dire che l'espressione è falsa
            · in modo analogo, dato che 1 OR $x = 1$, la valutazione cortocircuitata dell'OR prevede che se il primo argomento è vero, allora l'intera espressione è vera, senza dover valutare la seconda parte
    - sfruttando ciò, E viene valutato solo durante la prima iterazione del ciclo `do..while`, ovvero quando `prima_volta` è vero
    - nelle altre iterazioni, con `prima_volta` falso, la valutazione cortocircuitata evita di valutare E dentro l'`if`
    - **esercizio**: considerare un programma contenente un ciclo `while`, riscriverlo con un ciclo `do..while` come sopra e controllare che funzioni
    - **esercizio (difficile)**: effettuare la traduzione da `while` a `do..while` senza fare uso di variabili aggiuntive (suggerimento: usare `break`...)

- È possibile prendere un ciclo `while` e scrivere un ciclo `for` con la stessa semantica (ovvero, che fa la stessa cosa)
    - un generico ciclo `while` sarà scritto così
      `while (`E`)` I
    - un ciclo `for` con la stessa semantica è il seguente:
      `for (; `E`; )` I
    - tanto la parte di inizializzazione (che si presume eseguita prima del `while` stesso) quanto quella di iterazione (che si suppone contenuta in I) del `for` sono vuote
    - **esercizio**: considerare un programma contenente un ciclo `while`, riscriverlo con un ciclo `for` come sopra e controllare che funzioni

- È possibile prendere un ciclo `for` e scrivere un ciclo `do..while` con la stessa semantica (ovvero, che fa la stessa cosa)
    - provare a farlo per **esercizio**
    - **esercizio**: considerare un programma contenente un ciclo `while`, riscriverlo con un ciclo `do..while` come da precedente esercizio e controllare che funzioni

- È possibile prendere un ciclo `do..while` e scrivere un ciclo `while` con la stessa semantica (ovvero, che fa la stessa cosa)
    - provare a farlo per **esercizio**

  - **esercizio**: considerare un programma contenente un ciclo `do..while`, riscriverlo con un ciclo `while` come da precedente esercizio e controllare che funzioni

- È possibile prendere un ciclo `do..while` e scrivere un ciclo `for` con la stessa semantica (ovvero, che fa la stessa cosa)

  - provare a farlo per **esercizio**
  - **esercizio**: considerare un programma contenente un ciclo `do..while`, riscriverlo con un ciclo `for` come da precedente esercizio e controllare che funzioni

- **Esercizio (difficile)**: la presenza di `break`, `continue` e `return` può rendere non corrette alcune delle trasformazioni precedenti: quali? Come si può correggerle per tenerne conto?

- **Esercizio**: scrivere una semplice funzione con tipo di riturno `void` che mostri la differenza tra un `break` e una `return` (ovvero, se si sostituisce il `break` con una `return`, la semantica della funzione non è più la stessa)

- **Esercizio**: scrivere una semplice funzione con tipo di riturno `void` che mostri come un `break` e una `return` possano avere lo stesso effetto (ovvero, se si sostituisce il `break` con una `return`, la semantica della funzione è la stessa)

- È possibile riscrivere uno `switch` con una catena di `if`, ma non viceversa (e non solo perché si può testare la sola uguaglianza)

  - **esercizio**: scrivere un programma che legga un numero da tastiera e scriva (su righe diverse) se è divisibile per 2 (in questo caso dovrà dire che il numero è pari), 3, 5, 7, 11. È possibile farlo con uno `switch..case`?
  - **esercizio**: riscrivere la funzione di Figura 3 usando una catena di `if`. È necessario mettere anche gli `else` di tali `if`? Se non è necessario, rende almeno la funzione più efficiente?
  - **esercizio**: ricavare dal punto precedente la traduzione da `switch` a una catena di `if`, assumendo che ogni caso dello `switch` sia terminato da un `break`
  - **esercizio**: ricavare dal punto precedente la traduzione da `switch` a una catena di `if`