Programmazione 1

Lezione 4

Vincenzo Marra

vincenzo.marra@unimi.it

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

22 marzo 2017

Il flusso del controllo: `if-else`

- Sintassi.

```
if (espressione)
    istruzione_1
else
    istruzione_2
```

  Nota. La clausola `else` è facoltativa.

- Semantica.
  1. Si valuta *espressione*.
  2. Se il risultato è `vero`, si esegue $istruzione_1$, e si prosegue con la prima istruzione dopo l'istruzione `if-else`.
  3. Se il risultato è `falso`, ed è presente la clausola `else`, si esegue $istruzione_2$, e si prosegue con la prima istruzione dopo l'istruzione `if-else`.
  4. Se il risultato è `falso`, e non è presente la clausola `else`, si prosegue con la prima istruzione dopo l'istruzione `if`.

Il flusso del controllo: `else-if`

- Sintassi.

```
if (espressione_1)
    istruzione_1
else if (espressione_2)
    istruzione_2
    ⋮
else if (espressione_n)
    istruzione_n
else
    istruzione_{n+1}
```

Nota. L'ultima clausola `else` è facoltativa.

Il flusso del controllo: `else-if`

- Sintassi.

```
if (espressione_1)
    istruzione_1
else if (espressione_2)
    istruzione_2
    ⋮
else if (espressione_n)
    istruzione_n
else
    istruzione_{n+1}
```

  Nota. L'ultima clausola `else` è facoltativa.
- Semantica. Derivata da quella dell'istruzione `if-else`: infatti, il costrutto `else-if` non è una nuova istruzione, ma solo un caso particolare di applicazione di `if-else`.

Il flusso del controllo: else-if

- Nota Bene. Il codice appena visto è equivalente a:

```
if (espressione_1)
    istruzione_1
else
{
    if (espressione_2)
        istruzione_2
    else
    {
      ⋮
    }
}
```

Il flusso del controllo: la nozione di blocco

- Nella sintassi del C, un blocco è una porzione di codice racchiusa fra parentesi graffe: `{...}`.

Il flusso del controllo: la nozione di blocco

- Nella sintassi del C, un blocco è una porzione di codice racchiusa fra parentesi graffe: `{...}`.
- L'esempio più ovvio di blocco in C è il corpo di una funzione: `int main(void){...}`.

Il flusso del controllo: la nozione di blocco

- Nella sintassi del C, un blocco è una porzione di codice racchiusa fra parentesi graffe: `{...}`.
- L'esempio più ovvio di blocco in C è il corpo di una funzione: `int main(void){...}`.
- In generale, i blocchi si usano per raccogliere un gruppo di istruzioni in un tutto unico, di modo che il risultato sia sintatticamente equivalente a una singola istruzione.

Il flusso del controllo: la nozione di blocco

- Nella sintassi del C, un blocco è una porzione di codice racchiusa fra parentesi graffe: `{...}`.
- L'esempio più ovvio di blocco in C è il corpo di una funzione: `int main(void){...}`.
- In generale, i blocchi si usano per raccogliere un gruppo di istruzioni in un tutto unico, di modo che il risultato sia sintatticamente equivalente a una singola istruzione.
- Esempio leggero, senza blocco:

```
if ( san_val )
    prenota_rist();
```

Il flusso del controllo: la nozione di blocco

- Nella sintassi del C, un blocco è una porzione di codice racchiusa fra parentesi graffe: `{...}`.
- L'esempio più ovvio di blocco in C è il corpo di una funzione: `int main(void){...}`.
- In generale, i blocchi si usano per raccogliere un gruppo di istruzioni in un tutto unico, di modo che il risultato sia sintatticamente equivalente a una singola istruzione.
- Esempio impegnativo, con blocco (il corpo dell'istr. `if`):

```
if ( san_val )
{
    prenota_rist();
    compra_anel();
}
```

Il flusso del controllo: `switch-case`

- Sintassi.

```
switch (espressione)
{
case espr-cost_1 :   istruzioni_1
     ⋮
case espr-cost_n :   istruzioni_n
default:   istruzioni_{n+1}
}
```

Note. (1) Le *espr-cost*$_i$ sono espressioni *costanti* di tipo integrale, che devono essere tutte distinte. Per esempio, 3 o `'A'`; ma non, per esempio, variabili di tipo `char` o `int`. (2) La clausola `default` è opzionale, e può comparire ovunque nella lista dei casi.

Il flusso del controllo: `switch-case`

- Sintassi.

```
switch (espressione)
{
case espr-cost_1 :   istruzioni_1
     ⋮
case espr-cost_n :   istruzioni_n
default:   istruzioni_{n+1}
}
```

- Semantica. Si valuta *espressione*, ottenendo un valore intero V. Il controllo è spostato alla prima clausola `case` la cui espressione *espr-cost* vale V. Tale clausola, se esiste, è unica.

Il flusso del controllo: `switch-case`

- Sintassi.

```
switch (espressione)
{
case espr-cost_1 :   istruzioni_1
     ⋮
case espr-cost_n :   istruzioni_n
default:   istruzioni_{n+1}
}
```

- Semantica. Se nessuna clausola `case` ha espressione associata *espr-cost* di valore V, il controllo è spostato alla clausola `default`, se essa esiste. Se essa non esiste, nulla è eseguito oltre alla valutazione iniziale di *espressione*.

Il flusso del controllo: `switch-case`

- Sintassi.

```
switch (espressione)
{
case espr-cost_1 :   istruzioni_1
    ⋮
case espr-cost_n :   istruzioni_n
default:   istruzioni_{n+1}
}
```

- Semantica. Si noti bene: dire che il controllo è spostato ad una certa clausola significa che l'esecuzione proseguirà, da quel punto in poi, in modo sequenziale: non si saltano le clausole `case` successive. Si parla di esecuzione a cascata, in inglese *falling-through execution*.

Il flusso del controllo: break

- Il più delle volte, l'esecuzione a cascata dell'istruzione switch non corrisponde al flusso del controllo che si vorrebbe ottenere: è più comune voler interrompere l'esecuzione dell'intera istruzione `switch` *subito dopo* l'esecuzione del `case` di competenza.

Il flusso del controllo: break

- Il più delle volte, l'esecuzione a cascata dell'istruzione switch non corrisponde al flusso del controllo che si vorrebbe ottenere: è più comune voler interrompere l'esecuzione dell'intera istruzione `switch` *subito dopo* l'esecuzione del `case` di competenza.
- Per far ciò si usa l'istruzione `break`. Vi è una limitazione sintattica dei punti del codice in cui la si può usare:

Dallo standard C99, §6.8.6.3:

Constraints.
A break statement shall appear only in or as a switch body or loop body.

Semantics.
A break statement terminates execution of the smallest enclosing switch or iteration statement.

Il flusso del controllo: break

- Sintassi.

iterazione o `switch`

```
{
    ⋮
    break;
    ⋮
}
```

- Vincolo sintattico. L'istruzione `break` è ammessa solo all'interno del corpo (o come corpo stesso) di una iterazione o di un'istruzione `switch`.

Il flusso del controllo: break

- Sintassi.

iterazione o `switch`

```
{
    ⋮
    break;
    ⋮
}
```

- Semantica. L'esecuzione di `break` sposta il controllo del programma alla prima istruzione che segue il corpo della più vicina iterazione o istruzione `switch` che racchiude l'istruzione `break` stessa.

switch1.c

```c
#include <stdio.h>
int main(void)
{
        char car='B';
        switch (car)
        {
        case 'A': printf("Il carattere e' una A.\n");
        break;
        case 'B': printf("Il carattere e' una B.\n");
        break;
        case 'C': printf("Il carattere e' una C.\n");
        break;
        default: printf("Il carattere non e' ne' A, ne' B, ne' C.\n");
        break;
        }
        return 0;
}
```

(*Esecuzione dell'esempio.*)

switch2.c

```c
#include <stdio.h>
int main(void)
{
        int val=3;
        switch (val)
        {
        case 2: case 4: case 6:
        printf("Il numero e' in {1,2,3,4,5,6}, e pari.\n");
        break;
        case 1: case 3: case 5:
        printf("Il numero e' in {1,2,3,4,5,6}, e dispari.\n");
        break;
        default: printf("Il numero non e' in {1,2,3,4,5,6}\n");
        break;
        }
        return 0;
}
```

(*Esecuzione dell'esempio.*)

Il flusso del controllo: for

- Sintassi.

```
for (espr-iniz; espr-cond; espr-incr)
    istruzione
```

Il flusso del controllo: `for`

- Sintassi.

```
for (espr-iniz; espr-cond; espr-incr)
    istruzione
```

- Semantica. Il costrutto è equivalente[1] a:

```
espr-iniz;
while (espr-cond)
{
    istruzione
    espr-incr;
}
```

[1] Con l'eccezione dell'uso di `continue`, che discuteremo più avanti.

Il flusso del controllo: `for`

- Sintassi.

```
for (espr-iniz; espr-cond; espr-incr)
    istruzione
```

- Nota 1. Un sottoinsieme qualunque delle tre espressioni si può omettere, come pure l'istruzione. Se *espr-cond* è omessa, essa è assunta sempre `vera`. Quindi, `for(;;);` è un ciclo infinito come `while(1);`.

Il flusso del controllo: `for`

- Sintassi.

```
for (espr-iniz; espr-cond; espr-incr)
    istruzione
```

- Nota 1. Un sottoinsieme qualunque delle tre espressioni si può omettere, come pure l'istruzione. Se *espr-cond* è omessa, essa è assunta sempre `vera`. Quindi, `for(;;);` è un ciclo infinito come `while(1);`.
- Nota 2. L'espressione *espr-iniz* è spesso un assegnamento che inizializza una variabile indice. L'espressione *espr-incr* è spesso un incremento o un decremento della variabile indice.

postincremento.c

```c
#include <stdio.h>

int main(void)
{

        for (int i=0; i<10; i++)
                printf("%d\n",i);

        return 0;
}
```

L'istruzione `i++;` aumenta di uno il valore di `i`. È nota come istruzione di *postincremento*. Ne parleremo fra poco in questa stessa lezione.

(*Esecuzione dell'esempio.*)

postincremento.c

```c
#include <stdio.h>

int main(void)
{

        for (int i=0; i<10; i++)
                printf("%d\n",i);

        return 0;
}
```

Si noti che la variabile `i` è sia dichiarata che inizializzata nella *espr-iniz* del ciclo `for`. Ciò è permesso solo dallo standard C99 in poi.

(*Esecuzione dell'esempio.*)

postincremento2.c

```c
#include <stdio.h>

int main(void)
{
        int i;
        for (i=0; i<10; i++)
                printf("%d\n",i);

        return 0;
}
```

Negli standard precedenti è obbligatorio usare la codifica riporta qui sopra.

(*Esecuzione dell'esempio.*)

postincremento2.c

```c
#include <stdio.h>

int main(void)
{
        int i;
        for (i=0; i<10; i++)
                printf("%d\n",i);

        return 0;
}
```

Una differenza importante fra le due codifiche è che nella seconda la variabile `i` risulta definita anche dopo il ciclo `for`, mentre nella prima essa è definita solo all'interno del ciclo.

(*Esecuzione dell'esempio.*)

for.c

```c
/* Somma dei primi n numeri interi positivi con
la formula "di Gauss", e con il conto a forza
bruta tramite ciclo for */

#include <stdio.h>
int main(void)
{
        int n=100; // Precondizione: n>=1
        printf("(%d*(%d+1))/2=%d\n",n,n,(n*(n+1))/2);

        int r=0, i;
        for (i=1; i<=n; i++)
                r=r+i;

        printf("%d\n%d\n",r,i);
        return 0;
}
```

(Esecuzione dell'esempio.)

Uso while oppure for?

Uso `while` oppure `for`?

- `while` (o `do-while`) si usa tipicamente quando il numero di iterazioni da eseguire dipende da condizioni esterne non controllabili dal programmatore; per esempio, quando esso dipende dal fatto che l'utente prema un certo tasto invece che un altro: *leggi una riga fino a quando l'utente non scriva* `"E basta!"`.

Uso `while` oppure `for`?

- `while` (o `do-while`) si usa tipicamente quando il numero di iterazioni da eseguire dipende da condizioni esterne non controllabili dal programmatore; per esempio, quando esso dipende dal fatto che l'utente prema un certo tasto invece che un altro: *leggi una riga fino a quando l'utente non scriva* `"E basta!"`.
- `for` si usa tipicamente quando il numero di iterazioni da eseguire è direttamente controllabile dal programmatore; per esempio, quando esso è noto a priori come nel caso dell'esercizio pseudo-gaussiano che abbiamo appena visto: *somma i primi cento numeri*.

Incrementi e decrementi

- È comune l'uso dell'istruzione di *post-incremento*:

```
i++;
```
$(\star)$

dove `i` è, per esempio, una variabile di tipo `int`.

Incrementi e decrementi

- È comune l'uso dell'istruzione di *post-incremento*:

```
i++;
```
$(\star)$

  dove `i` è, per esempio, una variabile di tipo `int`.
- L'effetto dell'istruzione è di incrementare il valore di `i` di uno. Per esempio, se `i` vale 5, dopo l'istruzione $(\star)$ il valore di `i` è 6.

Incrementi e decrementi

- È comune l'uso dell'istruzione di *post-incremento*:

```
i++;
```
$(\star)$

  dove `i` è, per esempio, una variabile di tipo `int`.
- L'effetto dell'istruzione è di incrementare il valore di `i` di uno. Per esempio, se `i` vale 5, dopo l'istruzione $(\star)$ il valore di `i` è 6.
- È possibile anche l'uso dell'istruzione di *pre-incremento*:

```
++i;
```

  che ha lo stesso effetto di $(\star)$, ma con una importante differenza nella semantica.

- In una espressione composta in cui compaia il post-incremento:

  `i++`

  prima si usa il valore della variabile `i` e poi lo si incrementa di uno.

- In una espressione composta in cui compaia il post-incremento:

  `i++`

  prima si usa il valore della variabile `i` e poi lo si incrementa di uno.
- In una espressione composta in cui compaia il pre-incremento:

  `++i`

  prima si incrementa il valore della variabile di uno, e poi si usa il valore risultante.

- In una espressione composta in cui compaia il post-incremento:

  `i++`

  prima si usa il valore della variabile `i` e poi lo si incrementa di uno.
- In una espressione composta in cui compaia il pre-incremento:

  `++i`

  prima si incrementa il valore della variabile di uno, e poi si usa il valore risultante.
- Per esempio, se `i` vale 5, dopo l'istruzione:

  ```
  int x = i++;
  ```

  `x` vale 5. Invece, dopo l'istruzione

  ```
  int x = ++i;
  ```

  `x` vale 6. In entrambi i casi `i` varrà 6 dopo l'istruzione.

- In una espressione composta in cui compaia il post-incremento:

  `i++`

  prima si usa il valore della variabile `i` e poi lo si incrementa di uno.
- In una espressione composta in cui compaia il pre-incremento:

  `++i`

  prima si incrementa il valore della variabile di uno, e poi si usa il valore risultante.
- Per esempio, se `i` vale 5, dopo l'istruzione:

```
int x = i++;
```

  `x` vale 5. Invece, dopo l'istruzione

```
int x = ++i;
```

  `x` vale 6. In entrambi i casi `i` varrà 6 dopo l'istruzione.
- Considerazioni analoghe valgono per gli operatori di *pre-decremento*: `--i` e di *post-decremento*: `i--`.

- È diffuso l'uso degli operatori ++ e -- assieme all'istruzione `for`.

- È diffuso l'uso degli operatori ++ e -- assieme all'istruzione for.
- Supponiamo ad esempio di voler visualizzare i primi 100 numeri naturali in ordine crescente. Possiamo scrivere:

```
for(int i=1; i<=100; i++)
    printf("%d",i);
```

- Se invece volessimo visualizzare i primi 100 numeri naturali in ordine decrescente, potremmo scrivere:

```
for(int i=100; i>=1; i--)
    printf("%d",i);
```

- È diffuso l'uso degli operatori ++ e -- assieme all'istruzione for.
- Supponiamo ad esempio di voler visualizzare i primi 100 numeri naturali in ordine crescente. Possiamo scrivere:

```
for(int i=1; i<=100; i++)
    printf("%d",i);
```

- Se invece volessimo visualizzare i primi 100 numeri naturali in ordine decrescente, potremmo scrivere:

```
for(int i=100; i>=1; i--)
    printf("%d",i);
```

- Nel primo esempio è equivalente scrivere i=i+1, i++ o ++i; nel secondo è equivalente scrivere i=i-1, i-- o --i.

Altri operatori di assegnamento

- Abbiamo visto che = è l'operatore di *assegnamento*. Ve ne sono altri da esso derivati, fra i quali:

  `+=, -=, *=, /=, %=.`

Altri operatori di assegnamento

- Abbiamo visto che = è l'operatore di *assegnamento*. Ve ne sono altri da esso derivati, fra i quali:

  `+=, -=, *=, /=, %=.`

- Esempio:

```
int i=2;
i=i+3;  \\dopo questa istruzione i vale 5
```

- Si può anche scrivere:

```
int i=2;
i+=3;  \\dopo questa istruzione i vale 5
```

- C'è però una differenza fra la semantica di = e quella di +=.

- C'è però una differenza fra la semantica di = e quella di +=.
- `i=i+3` : Valuta il membro destro (ossia valuta `i`, valuta 3, e computa la somma dei risultati), *valuta il membro sinistro* (ossia valuta di nuovo `i`) e assegna il primo risultato al secondo.

- C'è però una differenza fra la semantica di = e quella di +=.
- `i=i+3` : Valuta il membro destro (ossia valuta `i`, valuta 3, e computa la somma dei risultati), *valuta il membro sinistro* (ossia valuta di nuovo `i`) e assegna il primo risultato al secondo.
- `i+=3` : Valuta il membro destro (ossia valuta 3), valuta il membro sinistro (ossia valuta `i`), e assegna la somma del primo e del secondo risultato al secondo.

- C'è però una differenza fra la semantica di = e quella di +=.
- `i=i+3` : Valuta il membro destro (ossia valuta `i`, valuta 3, e computa la somma dei risultati), *valuta il membro sinistro* (ossia valuta di nuovo `i`) e assegna il primo risultato al secondo.
- `i+=3` : Valuta il membro destro (ossia valuta 3), valuta il membro sinistro (ossia valuta `i`), e assegna la somma del primo e del secondo risultato al secondo.
- Nel primo caso `i` è valutata due volte, nel secondo caso una volta.

- C'è però una differenza fra la semantica di = e quella di +=.
- `i=i+3` : Valuta il membro destro (ossia valuta `i`, valuta 3, e computa la somma dei risultati), *valuta il membro sinistro* (ossia valuta di nuovo `i`) e assegna il primo risultato al secondo.
- `i+=3` : Valuta il membro destro (ossia valuta 3), valuta il membro sinistro (ossia valuta `i`), e assegna la somma del primo e del secondo risultato al secondo.
- Nel primo caso `i` è valutata due volte, nel secondo caso una volta.
- Negli esempi semplici come questo la differenza è trascurabile. In esempi più complessi la doppia valutazione può avere effetti collaterali importanti, e bisogna stare attenti.

- C'è però una differenza fra la semantica di = e quella di +=.
- `i=i+3` : Valuta il membro destro (ossia valuta `i`, valuta 3, e computa la somma dei risultati), *valuta il membro sinistro* (ossia valuta di nuovo `i`) e assegna il primo risultato al secondo.
- `i+=3` : Valuta il membro destro (ossia valuta 3), valuta il membro sinistro (ossia valuta `i`), e assegna la somma del primo e del secondo risultato al secondo.
- Nel primo caso `i` è valutata due volte, nel secondo caso una volta.
- Negli esempi semplici come questo la differenza è trascurabile. In esempi più complessi la doppia valutazione può avere effetti collaterali importanti, e bisogna stare attenti.
- Considerazioni analoghe valgono per gli altri operatori elencati sopra.

Esercizio in classe: Distanza euclidea

Esercizio.

Scrivere un programma che chieda all'utente di inserire le coordinate di due punti nel piano $\mathbb{R} \times \mathbb{R}$, e visualizzi poi la loro distanza nella metrica euclidea. Si usi il tipo `double`.

Suggerimento. Per estrarre la radice quadrata si usi la funzione `double sqrt(double)` contenuta nel file di intestazione `math.h` della libreria standard.

Nota. Quando si include il file `math.h`, in alcune implementazioni del compilatore C è necessario compilare con l'opzione `-lm` (per *link mathematical libraries*) per poter usare le funzioni aritmetiche definite nel file di intestazione.

distanza.c

```c
#include "Prog1/IO.h" //Include anche stdio.h
#include <math.h> //Per la funzione sqrt

int main(void)
{
        double px = leggi_double("Inserisci px: ");
        double py = leggi_double("Inserisci py: ");
        double qx = leggi_double("Inserisci qx: ");
        double qy = leggi_double("Inserisci qy: ");

        printf("La distanza fra (%g,%g) e (%g,%g) e' ",px,py,qx,qy);

        double d=sqrt( (px-qx)*(px-qx) + (py-qy)*(py-qy) );

        printf("%g.\n",d);

        return 0;
}
```

Esercizio in classe: Distanza euclidea iterata

Esercizio.

Modificare il programma scritto per risolvere l'esercizio precedente di modo che l'utente abbia la possibilità di iterare il calcolo della distanza per una nuova coppia di punti. Una volta finito un primo calcolo, il programma chiede all'utente se intende continuare, ricominciando da capo in caso di risposta affermativa e terminando l'esecuzione in caso contrario.

Suggerimento. Per decidere se continuare o no, il programma chiede all'utente di inserire una risposta `Sì/No`. Essa può essere acquisita leggendo un carattere dalla console con la funzione `char leggi_car()` della libreria `Prog1`. Per semplicità d'implementazione si può poi decidere di terminare l'esecuzione se il carattere letto è `'N'`, e continuare in qualunque altro caso.

distanza2.c

```c
#include "Prog1/IO.h" //Include anche stdio.h
#include <math.h> //Per la funzione sqrt

int main(void)
{
        char sc; //scelta utente: continua o no

        do
        {
                double px = leggi_double("Inserisci px: ");
                double py = leggi_double("Inserisci py: ");
                double qx = leggi_double("Inserisci qx: ");
                double qy = leggi_double("Inserisci qy: ");

                printf("La distanza fra (%g,%g) e (%g,%g) e' ", px,py,qx,qy);

                double d=sqrt( (px-qx)*(px-qx) + (py-qy)*(py-qy) );

                printf("%g.\n",d);

                printf("Continuare? (S/N) ");
                sc=leggi_car();
        } while (sc!='N'); //continua se sc non e' N

        return 0;
}
```

Esercizio in classe: La Trattoria

Esercizio.

Scrivere un programma che presenti all'utente il menu seguente.

1. Primo – 8.5 euro
2. Secondo – 10 euro
3. Contorno – 4 euro
4. Frutta – 3.5 euro
5. Conto

L'utente è libero di ordinare piatti dal menu in qualunque ordine e quantità. Dopo ogni ordinazione il programma scrive `Primo servito` (o `Secondo` ecc.) e visualizza nuovamente il menu. Se ciò che digita l'utente non corrisponde a una voce del menu, il programma visualizza un messaggio d'errore e torna al menu. Quando l'utente chiede il conto, il programma visualizza l'importo totale dovuto e termina.

trattoria.c

```c
#include "Prog1/IO.h"  //Per la funzione leggi_car e il tipo definito String
int main(void)
{
    char sc;                    // scelta dell'utente
    int itera=1;                // termina programma quando ==0
    double conto=0;             // importo totale dovuto
    String m="1. Primo\t8.5 e.\n2. Secondo\t10 e.\n3. Contorno\t4 e.\n4. Frutta\t3.5 e.\n5. Conto.\n";
    do{
        printf("%s> ",m);       // visualizza il menu e il prompt >
        sc=leggi_car();         // legge scelta dell'utente

        switch (sc){
        case '1':       printf("Primo servito.\n");
                        conto+=8.5;
        break;
        case '2':       printf("Secondo servito.\n");
                        conto+=10;
        break;
        case '3':       printf("Contorno servito.\n");
                        conto+=4;
        break;
        case '4':       printf("Frutta servita.\n");
                        conto+=3.5;
        break;
        case'5':        printf("Ecco il suo conto: %g euro. Grazie e arrivederci.\n",conto);
                        itera=0; //impostando itera=0 si ottiene l'uscita dal while
        break;
        default:        printf("Scelta inesistente.\n");
        break;
        }
    } while (itera); // continua fino a che itera==0
    return 0;
}
```