Flusso di controllo

Francesco Isgrò

- Le istruzioni di un programma normalmente sono eseguite secondo un flusso sequenziale
- Spesso è necessario alterare il flusso delle istruzioni
  - scelta fra differenti azioni
    - if
    - if-else
    - switch
  - ripetizione di un’azione
    - while
    - do while
    - for

- Il flusso è regolato in base a espressioni logiche che sono
  - TRUE
  - FALSE
- Fondamentali sono gli operatori
  - relazionali
  - di uguaglianza
  - logici

Operatori relazionali, di uguaglianza e logici

Operatori relazionali	minore maggiore minore o uguale maggiore o uguale	< > <= >=
Operatori di uguaglianza	uguale a diverso da	== !=
Operatori logici	negazione (unario) and logico or logico	! && ||

- Tutti binari, tranne !
- Operano su espressioni restituiscono un int (0 o 1)
- In C
  - FALSE rappresentato da un valore nullo
    - 0
    - 0.0
    - \0
  - TRUE rappresentato da un qualsiasi valore non nullo

Operatori relazionali e espressioni

- Prendono due espressioni come operandi
  - a < 3
  - a > b
  - -1.3 >= (2.0*x + 3.3)
- Espressioni non corrette
  - a =< b
  - a < = b
  - a>> b (corretto sintatticamente; altro significato)

- In matematica ha senso scrivere

  $a < b < c$

  per intendere che $(a < b) \wedge (b < c)$
- In C l'espressione è
  - sintatticamente corretta
  - interpretata come

    $(a < b) < c$

- a < b è equivalente a - b < 0
- a > b è equivalente a - b > 0
- a <= b è equivalente a - b <= 0
- a >= b è equivalente a - b >= 0
- Su molte macchine si valuta a – b

a - b	a > b	a >= b	a < b	a <= b
positivo	1	1	0	0
0	0	1	0	1
negativo	0	0	1	1

```
char c = ‘w’;
int i = 1, j = 2, k = -7;
double x = 7e^+33, y = 0.001;
```

espressione	espressione quivalente	valore
‘a’ + 1 < c	(‘a’ + 1) < c	
-i – 5 * j >= k + 1	((-i) – (5 * j)) >= (k + 1)	
3 < j < 5	(3 < j) < 5	
x – 3.333 <= x + y	(x – 3.333) <= (x + y)	
x < x + y	x < (x + y)	

```
char c = ‘w’;
int i = 1, j = 2, k = -7;
double x = 7e^+33, y = 0.001;
```

espressione	espressione quivalente	valore
‘a’ + 1 < c	(‘a’ + 1) < c	1
-i – 5 * j >= k + 1	((-i) – (5 * j)) >= (k + 1)	0
3 < j < 5	(3 < j) < 5	1
x – 3.333 <= x + y	(x – 3.333) <= (x + y)	1
x < x + y	x < (x + y)	0

- I numeri floating point non hanno precisione infinita
- A volte questo può causare risultati inaspettati
- In matematica la relazione

  $x < x + y$

  è equivalente a

  $y > 0$

- Consideriamo il caso in cui
  - x variabile reale molto grande
  - y variabile reale molto piccola
- Ad esempio

```
x = 7e+33;
y = 0.001;
x < x + y
```

```
#include <stdio.h>

int main ()
{
	double x = 7e+33;
	double y = 0.001;

	if (x < x + y) printf("x < x + y\n");
	if (x > x + y) printf("x > x + y\n");
	if (x == x + y) printf("x = x + y\n");
}
```

- Il programma stampa
  - x = x + y
- L’incremento rispetto ad x dato da y è troppo piccolo
- La precisione non è abbastanza per rendere la variazione
- La rappresentazione di x rimane invariata

Operatori di uguaglianza ed espressioni

- Gli operatori di uguaglianza == e != sono operatori binary che agiscono sulle espressioni
- Ad esempio
  - c == ‘A’;
  - k != -2;
  - x + y == 3 * z – 7;

- Non sono esempi corretti
  - a = b; //istruzione di assegnamento
  - a = = b – 1; // spazio non consentito
  - (x + y) =! 44; //equivalente a (x + y) = (!44); errore

- Una espressione

  expr1 == expr2

  è TRUE o FALSE
- Una espressione equivalente è

  expr1 – epr2 == 0
- L’operatore è solitamente implementato in quest’ultima maniera

- L’espressione
  - expr1 != expr2
- è implementata analogamente come
  - expr1 – expr2 != 0

expr1 – expr2	expr1 == expr2	expr1 != expr2
zero	1	0
non zero	0	1

```
int i, j, k;

i = 1;
j = 2;
k = 3;
```

espressione	espressione equivalente	valore
i == j	i - j == 0	0
i != j	i – j != 0	1
i+j+k == -2*-k	((i+j)+k) == ((-2)*(-k))	1

- Un errore molto comune è confondere le due espressioni
  - a == b
  - a = b
- Ad esempio

  ```
  if (a = b)
  ```

  invece di

  ```
  if (a == b)
  ```

- La prima è TRUE o FALSE a seconda del solo valore di b

Operatori logici ed espressioni

- L'operatore logico di negazione ! è unario
- I due operatori di
  - AND &&
  - OR ||

  sono binari
- Se un’espressione ha valore 0 la sua negata ha valore 1 e viceversa

- Alcuni esempi corretti

  ```
  !a
  !(x + 7.7)
  !(a < b || c < d)
  ```

- Esempi non corretti

  ```
  a!          //ordine sbagliato
  a  !=  b    //operatore diverso. ! non usato come
              //operatore unario
  ```

- Se A è un’affermazione in logica si ha
  - not(not A) = A
- In C si possono avere comportamenti diversi
- Consideriamo
  - !!5

- Se A è un’affermazione in logica si ha
  - not(not A) = A
- In C si possono avere comportamenti leggermente diversi
- Consideriamo
  - !!5
  - L’espressione ha valore 1

```
char    c = ‘A’;
int     i = 7, j = 7;
double  x = 0.0, y = 2.3;
```

espressione	espressione equivalente	valore
!c	!c	
!(i - j)	!(i - j)	
!i - j	(!i) - j	
!!(x + y)	!(!(x + y))	
!x * !!y	(!x) * (!(!y))	

```
char    c = ‘A’;
int     i = 7, j = 7;
double  x = 0.0, y = 2.3;
```

espressione	espressione equivalente	valore
!c	!c	0
!(i - j)	!(i - j)	1
!i - j	(!i) - j	-7
!!(x + y)	!(!(x + y))	1
!x * !!y	(!x) * (!(!y))	1

- Gli operatori binari && e || agiscono su espressioni e ritornano i valori interi 1 (TRUE) o 0 (FALSE)
- Alcuni esempi

```
a && b
a || b
!(a < b) && c
3 && (-2 * a +7)
```

- Esempi non corretti

```
a &&          // manca uno degli operandi
a |  | b      // spazi non permessi
&b            // indica l'indirizzo di b
```

Tabella di verità			
expr1	expr2	expr1 && expr2	expr1 || expr2
zero	zero	0	0
zero	non zero	0	1
non zero	zero	0	1
non zero	non zero	1	1

- L’operatore && ha la precedenza su ||
- Entrambi gli operatori hanno priorità bassa rispetto agli operatori unari e relazionali
- L'associatività va da sinistra a destra

```
char    c = 'B';
int     i = 3, j = 3, k = 3;
double  x = 0.0, y = 2.3;
```

espressione	espressione equivalente	valore
i && j && k		
x || i && j - 3		
i < j && x < y		
i < j || x < y		
'A' <= c && c <= 'Z'		
c – 1 == 'A' || c + 1 == 'Z'		

```
char    c = 'B';
int     i = 3, j = 3, k = 3;
double  x = 0.0, y = 2.3;
```

espressione	espressione equivalente	valore
i && j && k	(i && j) && k	
x || i && j - 3	x || (i && (j - 3))	
i < j && x < y	(i < j) && (x < y)	
i < j || x < y	(i < j) || (x < y)	
'A' <= c && c <= 'Z'	('A' <= c) && (c <= 'Z')	
c – 1 == 'A' || c + 1 == 'Z'	((c – 1) == 'A') || ((c + 1) == 'Z')	

```
char    c = 'B';
int     i = 3, j = 3, k = 3;
double  x = 0.0, y = 2.3;
```

espressione	espressione equivalente	valore
i && j && k	(i && j) && k	1
x || i && j - 3	x || (i && (j - 3))	0
i < j && x < y	(i < j) && (x < y)	0
i < j || x < y	(i < j) || (x < y)	1
'A' <= c && c <= 'Z'	('A' <= c) && (c <= 'Z')	1
c – 1 == 'A' || c + 1 == 'Z'	((c – 1) == 'A') || ((c + 1) == 'Z')	1

Valutazione short circuit

- La valutazione delle espressioni con operandi && e || si blocca appena il risultato è noto
- Nella valutazione dell'espressione

  expr1 && expr2

  - se expr1 è vera allora valuta anche expr2
  - se expr1 è falsa non è necessario valutare expr2

- Nella valutazione dell’espressione
  - expr1 || expr2
  - se expr1 è falsa allora valuta anche expr2
  - se expr1 è vera allora non è necessario valutare expr2

```
#include <stdio.h>
int main()
{
        int a , b ;

        a = 0;
        b = 0;
        while ( a < 3 && ++b ) {
                a++;
        }
        printf ( " a =% d b =% d \ n " , a , b );
}
```

```
#include <stdio.h>
int main()
{
        int a , b ;

        a = 0;
        b = 0;
        while ( a < 3 && ++b ) {
                a++;
        }
        printf ( " a =% d b =% d \ n " , a , b );
}
```

a = 3 b = 3

```
#include <stdio.h>
int main()
{
	int a , b ;

	a = 0;
	b = 0;
	while ( ++b && a < 3) {
		a++;
	}
	printf ( " a =% d b =% d \ n " , a , b );
}
```

```
#include <stdio.h>
int main()
{
        int a , b ;

        a = 0;
        b = 0;
        while ( ++b && a < 3 ) {
                a++;
        }
        printf ( " a =% d b =% d \ n " , a , b );
}
```

a = 3 b = 4

if e if-else

- La forma generale dell'if è

```
if ( expr )
    statement;
statementSucc;
```

- Se expr è non-zero (TRUE) allora viene eseguita statement
- Se expr è zero (FALSE) allora statement non viene eseguita e il controllo passa all'istruzione successiva statementSucc

- expr può essere qualsiasi espressioni ma solitamente è del tipo logico, relazionale o di uguaglianza
  - if (grade >= 90)

    printf (" Congratulations \ n ");

    printf (" Your grade is %d .\ n " , grade);
  - if (c == ’ ’) {

    ++ blank_cnt ;

    printf (" found another blank ");

    }

- è preferibile usare un’istruzione composta sotto il controllo di un solo if
  - Non consigliato

```
if ( j < k )
    min = j ;
if (j < k )
    printf ( " j is smaller than k \ n " );
```

- è preferibile usare un’istruzione composta sotto il controllo di un solo if
  - Consigliato

```
if ( j < k ) {
    min = j ;
    printf ( " j is smaller than k \ n " );
}
```

- è preferibile usare un’istruzione composta sotto il controllo di un solo if
  - Non consigliato

```
if ( j < k )
    min = j ;
```

- è preferibile usare un’istruzione composta sotto il controllo di un solo if
  - Consigliato

```
if ( j < k ) {
    min = j ;
}
```

- La forma generale dell'if-else è

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

- Se expr è non-zero viene eseguito statement1
- Se expr è zero viene eseguito statement2

- Uno statement if può a sua volta essere usato all'interno di un if . Analogamente per if-else
- Consideriamo il blocco di codice

```
if ( a == 1)
    if ( b == 2)
        printf ( " * * *\ n " );
```

- Il blocco è della forma

```
if ( a == 1)
    statement
```

dove statement è

```
if ( b == 2)
    printf ( " ***\ n " );
```

Esempio 1

```
if ( a == 1)
        if ( b == 2)
                printf ( " ***\ n " );
        else
                printf ( " ###\ n " );
```

Esempio 2

```
if ( a == 1)
        if ( b == 2)
                printf ( " ***\ n " );
else
        printf ( " ###\ n " );
```

C’è differenza fra i due blocchi?

- La formattazione del testo suggerisce due semantiche differenti
- Per il compilatore la formattazione non ha significato
- L’ else è associato all’ if più vicino.
- Per chiarezza è preferibile che la formattazione rispecchi la semantica

Esempio 1

```
if ( a == 1)
        if ( b == 2)
                printf ( " ***\ n " );
        else
                printf ( " ###\ n " );
```

Esempio 2

```
if ( a == 1)
        if ( b == 2)
                printf ( " ***\ n " );
else
        printf ( " ###\ n " );
```

Il while

- La forma generale del costrutto while è

```
while ( expr )
    statement1
statement2
```

- Viene valutata expr
- Se expr è non-zero (TRUE)
  - viene eseguito statement1
  - il controllo ritorna all'inizio del loop while

Il while

- La forma generale del costrutto while è

```
while ( expr )
    statement1
statement2
```

- Viene valutata expr
- Se expr è zero (FALSE) il controllo passa a statement2
- Essenzialmente statement1 è eseguito ripetutamente fino a quando expr diventa falsa

Attenti ai loop infiniti

```
printf ( " Input a positive integer :   “);
scanf ( " % d " , & n );
while ( - - n ) {
        .....  // do something
}
```

Attenti ai loop infiniti

```
printf ( " Input a positive integer :   “);
scanf ( " % d " , & n );
while ( - - n ) {
        .....  // do something
}
```

- Si assume che sia inserito un intero positivo
- Il corpo del while deve essere eseguito ripetutamente finché --n diventa 0
- Cosa succede se viene dato in input un numero non positivo?

- Si può controllare meglio il loop

```
printf ( " Input a positive integer :   “);
scanf ( " % d " , & n );
while ( - - n > 0) {
        …….  // do something
}
```

- Si può controllare l’input

```
printf ( " Input a positive integer :   “);
scanf ( " % d " , & n );
if (n <= 0) {
        errore();
}
while ( - - n > 0) {
        …….  // do something
}
```

- A volte può essere necessario che il corpo del while sia uno statement vuoto

```
while (( c = getchar ()) == ' ');
```

  - Permette di saltare tutti i caratteri blank consecutivi
- Per rendere evidente che è intenzionale si suggerisce la forma

```
while (( c = getchar ()) == ' ')
    ;
```

```c
/* Count blanks, digits, letters, newlines, and others. */

#include <stdio.h>

int main(void)
{
  int   blank_cnt = 0, c, digit_cnt = 0,
        letter_cnt = 0, nl_cnt = 0, other_cnt = 0;

  while ((c = getchar()) != EOF)     /* braces not necessary */
    if (c == ' ')
      ++blank_cnt;
    else if (c >= '0' && c <= '9')
      ++digit_cnt;
    else if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z')
      ++letter_cnt;
    else if (c == '\n')
      ++nl_cnt;
    else
      ++other_cnt;

  printf("%10s%10s%10s%10s%10s%10s\n\n",
    "blanks", "digits", "letters", "lines", "others", "total");
  printf("%10d%10d%10d%10d%10d%10d\n\n",
    blank_cnt, digit_cnt, letter_cnt, nl_cnt, other_cnt,
    blank_cnt + digit_cnt + letter_cnt + nl_cnt + other_cnt);
  return 0;
}
```

```
./a.out < cnt_char.c
    blanks    digits   letters     lines    others     total

       208        32       361        28       180       809
```

Istruzione for

- La forma è

```
for (expr1; expr2; expr3)
    statementFor;
statementSucc;
```

Istruzione for

- La forma è

```
for (expr1; expr2; expr3)
    statementFor;
statementSucc;
```

- Equivalente a

```
expr1;
while (expr2) {
    statementFor;
    expr3;
}
statementSucc;
```

Espressioni relazionali o uguaglianza

- A volte è preferibile è meglio usare espressioni relazionali invece di espressioni di uguaglianza
- Ciò permette di ottenere codice più robusto
- Per espressioni di tipo `float` o `double` un test di uguaglianza può essere ad di fuori della precisione della macchina

```
/* A test */

#include <stdio.h>

int main(void)
{
   int      cnt = 0;
   double   sum = 0.0, x;

   for (x = 0.0; x != 9.9; x += 0.1) {
      sum += x;
      printf("cnt = %5d\n", ++cnt);
   }
   printf("sum = %f\n", sum);
   return 0;
}
```

```
/* A test that fails. */

#include <stdio.h>

int main(void)
{
   int      cnt = 0;
   double   sum = 0.0, x;

   for (x = 0.0; x != 9.9; x += 0.1) {   /* trouble! */
      sum += x;
      printf("cnt = %5d\n", ++cnt);
   }
   printf("sum = %f\n", sum);
   return 0;
}
```

Successione di Fibonacci

- Definita ricorsivamente come
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_{n+1} = f_n + f_{n-1}$
- I primi elementi della successione sono
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

- La successione dei quozienti di Fibonacci è definita come
  - $q_n = f_n / f_{n-1}$
- Si dimostra che converge alla sezione aurea

$$q_n \rightarrow 1 + \sqrt{5} \approx 1.6180339887498948 4 8$$

- Vediamo un programma che stampa
  - i numeri di Fibonacci
  - i quozienti di Fibonacci