Capitolo 5

Istruzioni di selezione

Istruzioni

- Fino ad ora abbiamo usato l'istruzione `return` e istruzioni-espressioni
- La maggior parte delle rimanenti istruzioni in C possono essere classificate in 3 categorie:
  - ***Istruzioni di selezione:*** `if` e `switch`
  - ***Istruzioni di iterazione:*** `while`, `do`, e `for`
  - ***Istruzioni di salto:*** `break`, `continue`, e `goto`. (`return` appartiene a questa categoria.)
- Altri tipi di istruzioni:
  - Istruzioni composte
  - Istruzione nulla

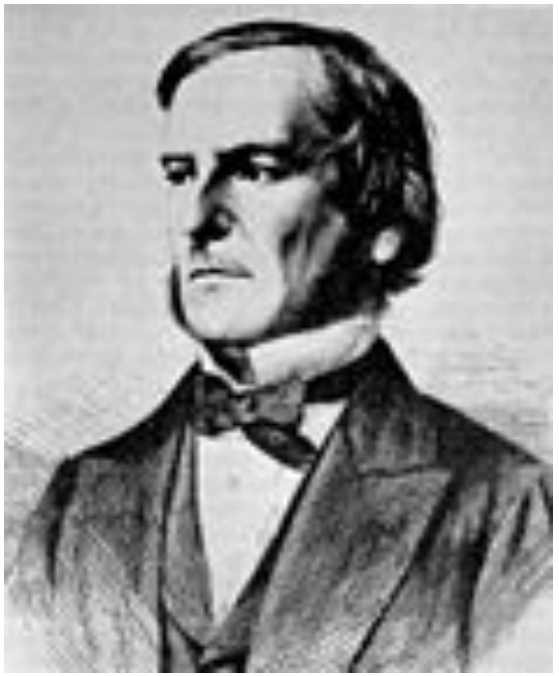
Espressioni logiche

- Le istruzioni di selezione del C devono verificare se una certa condizione è "**vera**" o "**falsa**"
- Per esempio, l'istruzione `if` potrebbe dover controllare l'espressione `i < j`; il valore "vero" indicherebbe che `i` è minore di `j`
- In molti linguaggi di programmazione una espressione come `i < j` avrebbe un "tipo" speciale, "Booleano" o "logico"
- In C un confronto come `i < j` dà come risultato un intero: 0 (falso) oppure 1 (vero)

Booleano...

Nel 1854 pubblicò la sua opera più importante,
An Investigation of the Laws of Thought,
indirizzata alle leggi del pensiero …

Con questo lavoro fondò la teoria delle algebre di Boole (o, semplicemente, "algebra booleana").



George Boole

È stata la base (grazie a Claude Shannon, che ha riconosciuto la coincidenza tra il funzionamento dei circuiti commutatori e la logica proposizionale), degli studi sui circuiti elettronici e sulla commutazione, e ha costituito un passo importante verso la concezione dei moderni computer.

Operatori relazionali

- Gli ***operatori relazionali*** in C sono

  `<` minore di
  `>` maggiore di
  `<=` minore o uguale a
  `>=` maggiore o uguale a

- Questi operatori producono 0 (falso) oppure 1 (vero) quando usati nelle espressioni
- Gli operatori relazionali possono confrontare interi `int` o numeri a virgola mobile `float`
- È possibile mischiare operandi di tipo diverso (`int` e `float`)

Operatori relazionali

- La precedenza degli operatori relazionali è più bassa di quella degli operatori aritmetici
  - Per esempio
  - `i + j < k - 1` equivale a `(i + j) < (k - 1)`
  - Quanto vale `i + (j < k) - 1` ?
- Gli operatori relazionali sono associativi a sinistra.

Operatori relazionali

- L'espressione

  `i < j < k`

  è valida, ma **non** controlla se `j` è compreso fra `i` e `k`
- Poichè l'operatore < è associativo a sinistra questa espressione è equivalente a

  `(i < j) < k`

  Il valore 1 o 0 prodotto da `i < j` viene confrontato con `k`

Operatori di uguaglianza

- Il C fornisce due ***operatori di uguaglianza:***

  `==` uguale a
  `!=` diverso da

- Questi operatori sono associativi a sinistra e producono 0 (falso) oppure 1 (vero)
- Gli operatori di uguaglianza hanno precedenza ancora più bassa degli operatori relazionali

- L'espressione

  ```
  i < j == j < k
  ```

  è equivalente all'espressione

  ```
  (i < j) == (j < k)
  ```

Operatori Logici

- Espressioni logiche più complicate possono essere costruite usando gli ***operatori logici:***

  `!` negazione logica
  `&&` *and* logico
  `||` *or* logico

- L'operatore `!` è unario, mentre `&&` e `||` sono binari
- Producono 0 o 1 come risultato
- Trattano un operando diverso da zero come "vero" e un operando pari a zero come "falso"

Operatori Logici

- Operazioni logiche:

  !*expr* vale 1 se *expr* vale 0

  expr1 && *expr2* vale 1 se *expr1* e *expr2* sono **entrambe** diverse da zero

  expr1 || *expr2* vale 1 se **almeno** una fra *expr1* e *expr2* ha un valore diverso da 0

- In tutti gli altri casi il risultato è 0.

Operatori Logici

- Sia `&&` che `||` eseguono una valutazione detta a "circuito breve" ("short-circuit"):
  - Prima valutano l'operando sinistro e poi quello destro
- Se il valore finale può essere dedotto direttamente dall'operando sinistro, l'operando destro non viene valutato.
  - Esempio:

    ```
    (i != 0) && (j / i > 0)
    ```

    l'espressione `(i != 0)` viene valutata. Se `i` è diverso da 0, allora `(j / i > 0)` viene valutata
  - Se `i` è uguale 0, non c'è bisogno di valutare `(j / i > 0)`. Senza la valutazione a circuito breve avremmo causato una divisione per 0

Operatori Logici

- A causa della valutazione a circuito breve di `&&` e `||`, gli effetti collaterali potrebbero non verificarsi sempre
- Esempio:

  ```
  i > 0 && ++j > 0
  ```

  Se `i > 0` è falso, allora `++j > 0` non viene valutato e quindi `j` non viene incrementata
- Il problema può essere eliminato cambiando l'espressione in `++j > 0 && i > 0`
- Oppure, meglio ancora, incrementando `j` separatamente

Operatori Logici

- L'operatore `!` ha la stessa precedenza degli operatori unari di addizione e sottrazione
- La precedenza di `&&` e `||` è più bassa di quella degli operatori relazionali e di uguaglianza

Per esempio,
`i < j && k == m` equivale a `(i < j) && (k == m)`.

- L'operatore `!` è associativo a destra, mentre `&&` e `||` sono associativi a sinistra

L'istruzione `if`

- L'istruzione `if` permette ad un programma di scegliere fra due alternative
- Nella sua forma più semplice, l'istruzione `if` è
- `if (` *expression* `)` *statement*
- Quando un `if` viene eseguito, *expression* viene valutata; se il suo valore è "vero" allora l'istruzione *statement* viene eseguita
- Esempio:

```
if (line_num == MAX_LINES)
  line_num = 0;
```

L'istruzione `if`

- Confondere == (uguaglianza) con = (assegnamento) è forse l'errore più comune nella programmazione in C
- L'istruzione

  ```
  if (i == 0) …
  ```

  controlla se `i` è uguale a 0
- L'istruzione

  ```
  if (i = 0) …
  ```

  assegna 0 ad `i`, e successivamente controlla se il valore risultante è 0 o diverso da 0

L'istruzione `if`

- Spesso l'istruzione `if` controlla se una variabile è in un determinato intervallo
- Per controllare se `0` ≤ `i` < `n`:

```
if (0 <= i && i < n) …
```

- Per controllare la condizione opposta (cioè se `i` è fuori dall'intervallo):

```
if (i < 0 || i >= n) …
```

oppure

```
if !(0 <= i && i < n) …
```

Istruzioni composte

- Si noti che il "corpo" dell'istruzione `if` è formato da una **singola** istruzione *statement*:

  `if` `(` *expression* `)` *statement*

- Per far sì che il corpo contenga due o più istruzioni si deve usare un'istruzione composta
- Un'istruzione composta è un blocco di istruzioni delimitato da parentesi graffe

  `{` *statements* `}`

- L'uso di parentesi graffe dice al compilatore di trattare tutte le istruzioni del blocco come una singola istruzione

Istruzioni composte

- Esempio:

```
{ line_num = 0; page_num++; }
```

- È prassi comune scrivere un blocco di istruzioni mettendo una istruzione per linea:

```
{
  line_num = 0;
  page_num++;
}
```

- Si noti che ogni singola istruzione del blocco termina con il punto e virgola, ma il blocco stesso **non** ha un punto e virgola finale

Istruzioni composte

- Esempio di istruzione composta in un `if`

```
if (line_num == MAX_LINES) {
  line_num = 0;
  page_num++;
}
```

- Le istruzioni composte sono molto comuni in cicli ed altri casi in cui il C richiede una singola istruzione

La clausola `else`

- Un'istruzione `if` può avere una clausola `else`

`if ( `*expression*` ) `*statement*` else `*statement*

- L'istruzione che segue la parola chiave `else` viene eseguita quando il valore di *expression* è 0
- Esempio:

```
if (i > j)
  max = i;
else
  max = j;
```

La clausola `else`

- Dove scrivere la clausola `else`?
- Molti programmatori C allineano l'`else` all'`if`
- Le istruzioni interne sono solitamente indentate ma se sono brevi possono essere messe sulla stessa linea dell'`if` e dell'`else`:

```
if (i > j) max = i;
else max = j;
```

La clausola `else`

- Un `if` può apparire all'interno di un altro `if`:

```
if (i > j)
  if (i > k)
    max = i;
  else
    max = k;
else
  if (j > k)
    max = j;
  else
    max = k;
```

- Allineare ogni `else` con il corrispondente `if` rende il programma molto più leggibile:

```
if (i > j)if (i > k) max = i; else max = k;
else if (j > k) max = j; else max = k;
```

La clausola `else`

- Per evitare confusione non esitate ad usare le parentesi graffe anche quando non sono necessarie:

```
if (i > j) {
   if (i > k)
      max = i;
   else
      max = k;
} else {
   if (j > k)
      max = j;
   else
      max = k;
}
```

La clausola `else`

- Alcuni programmatori mettono sempre le parentesi graffe in un'istruzione `if`:

```
if (i > j) {
  if (i > k) {
    max = i;
  } else {
    max = k;
  }
} else {
  if (j > k) {
    max = j;
  } else {
    max = k;
  }
}
```

La clausola `else`

- I vantaggi nel fare ciò sono:
  - Facilita la modifica: per aggiungere ulteriori istruzioni non bisogna preoccuparsi delle parentesi graffe (già ci sono)
  - Aiuta ad evitare errori dovuti alla mancanza di parentesi graffe quando sono necessarie

Istruzioni `if` a cascata

- Delle istruzioni `if` "a cascata" sono molto utili per controllare una serie di condizioni fermandosi alla prima che risulta vera

- Esempio:

```
if (n < 0)
  printf("n is less than 0\n");
else
  if (n == 0)
    printf("n is equal to 0\n");
  else
    printf("n is greater than 0\n");
```

Istruzioni `if` a cascata

- In questo caso, sebbene il secondo `if` sia di fatto una parte interna del primo, è prassi comune non usare l'indentazione

- Tutti gli else vengono allineati con il primo `if`:

```
if (n < 0)
  printf("n is less than 0\n");
else if (n == 0)
  printf("n is equal to 0\n");
else
  printf("n is greater than 0\n");
```

Istruzioni `if` a cascata

- Questa forma elimina problemi di eccessiva indentazione nel caso in cui ci siano molti `if` a cascata:

```
if ( expression )
   statement
else if ( expression )
   statement
...
else if ( expression )
   statement
else
   statement
```

Programma: commissione broker

- Quando si comprano delle azioni attraverso un broker, il pagamento del broker dipende dal valore delle azioni
- Supponiamo che un broker si faccia pagare secondo la seguente tabella:

Valore transazione	*Commissione*
< $2,500	$30 + 1.7%
$2,500–$6,250	$56 + 0.66%
$6,250–$20,000	$76 + 0.34%
$20,000–$50,000	$100 + 0.22%
$50,000–$500,000	$155 + 0.11%
> $500,000	$255 + 0.09%

- Inoltre c'è un valore minimo della commissione di $39

Programma: commissione broker

- Il programma `broker.c` chiede all'utente di immettere l'ammontare della transazione e calcola la commissione:

```
Enter value of trade: 30000
Commission: $166.00
```

- Il cuore del programma è una serie di `if` a cascata che controllano l'intervallo nel quale ricade la transazione

broker.c

```
/* Calculates a broker's commission */

#include <stdio.h>

int main(void)
{
  float commission, value;

  printf("Enter value of trade: ");
  scanf("%f", &value);

  if (value < 2500.00f)
    commission = 30.00f + .017f * value;
  else if (value < 6250.00f)
    commission = 56.00f + .0066f * value;
  else if (value < 20000.00f)
    commission = 76.00f + .0034f * value;
  else if (value < 50000.00f)
    commission = 100.00f + .0022f * value;
  else if (value < 500000.00f)
    commission = 155.00f + .0011f * value;
  else
    commission = 255.00f + .0009f * value;
```

```
  if (commission < 39.00f)
    commission = 39.00f;

  printf("Commission: $%.2f\n", commission);

  return 0;
}
```

Il problema dell' `else` "pendente"

- Quando si usano degli `if` a cascata si può avere il problema dell'`else` pendente:

```
if (y != 0)
  if (x != 0)
    result = x / y;
else
  printf("Error: y is equal to 0\n");
```

- L'indentazione sembra indicare che l'`else` fa riferimento all'`if` esterno
- Tuttavia il C segue la regola che un `else` fa riferimento **al più vicino** `if` che non è già stato "abbinato" con un altro `else`.

Il problema dell' `else` "pendente"

- Quindi l'indentazione corretta è:

```
if (y != 0)
  if (x != 0)
    result = x / y;
  else
    printf("Error: y is equal to 0\n");
```

- che però non è il comportamento desiderato

Il problema dell' `else` "pendente"

- Per rendere l'`else` parte del primo `if` occorre usare le parentesi graffe:

```
if (y != 0) {
  if (x != 0)
    result = x / y;
} else
  printf("Error: y is equal to 0\n");
```

- Se avessimo adottato la tecnica di mettere sempre le parentesi graffe non avremmo avuto il problema

Espressioni condizionali

- L'***operatore condizionale*** permette di creare un'espressione che produce uno fra due valori in funzione di una condizione
- L'operatore condizionale è specificato da 2 simboli, `?` e `:`, che devono essere usati insieme nel seguente modo

  expr1 `?` *expr2* `:` *expr3*
- Gli operandi possono essere di qualsiasi tipo
- Un'espressione di questo tipo è detta ***espressione condizionale.***

Espressioni condizionali

- L'operatore condizionale richiede 3 operandi, quindi è un operatore ***ternario***
- L'espressione condizionale *expr1* `?` *expr2* `:` *expr3* dovrebbe essere letta come "if *expr1* then *expr2* else *expr3*."
- L'espressione condizionale viene valutata così:
  - Prima si valuta *expr1*
  - Se il suo valore è diverso da 0 (vero) allora viene valutata *expr2* ed il valore risultante è il valore di tutta l'espressione condizionale
  - Se il valore di *expr1* è 0 (falso) allora il valore di tutta l'espressione condizionale è il valore di *expr3*

Espressioni condizionali

- Esempio:

```
int i, j, k;

i = 1;
j = 2;
k = i > j ? i : j;              /* k is now 2 */
k = (i >= 0 ? i : 0) + j;       /* k is now 3 */
```

- Le parentesi sono necessarie perchè la precedenza dell'operatore condizionale è più bassa di quella di tutti gli altri operatori visti finora con l'eccezione dell'operatore di assegnamento

Espressioni condizionali

- Le espressioni condizionali permettono di scrivere codice conciso, ma rendono la lettura più difficile
  - Usarle con parsimonia

- Sono usate nelle macro

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

- Nelle istruzioni `return`:

```
return i > j ? i : j;
```

Espressioni condizionali

- Anche nella `printf` possono essere comode

- Invece di

```
if (i > j)
  printf("%d\n", i);
else
  printf("%d\n", j);
```

possiamo scrivere

```
printf("%d\n", i > j ? i : j);
```

L'istruzione `switch`

- Un `if` a cascata viene usato spesso per controllare se una variabile assume un particolare valore :

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```

L'istruzione `switch`

- L'istruzione `switch` permette di fare proprio questo:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

L'istruzione `switch`

- L'istruzione `switch` è più facile da leggere (rispetto all'`if` a cascata)
- Inoltre, l'istruzione `switch` è spesso più veloce di un `if` a cascata
- La forma più comune di `switch` è:

```
switch ( expression ) {
   case constant-expression : statements
   ...
   case constant-expression : statements
   default : statements
}
```

L'istruzione `switch`

- La parola chiave `switch` deve essere seguita da un'espressione che una volta valutata è un intero
  - Tale espressione viene detta ***espressione di controllo***
  - È racchiusa tra parentesi

- Il tipo `char` è trattato come un `int` quindi può essere usato come espressione di controllo in uno `switch`

- Invece `float` e stringhe **non** possono essere usate

L'istruzione `switch`

- Ogni "diramazione" dello `switch`, inizia con la parola chiave `case`

  `case` *constant-expression* `:`

- L'***espressione costante*** è simile ad una normale espressione, ma non può contenere variabili o chiamate a funzioni
  - `5` è un'espressione costante, ed anche `5 + 10` è un'espressione costante, ma `n + 10` non lo è
    - a meno che `n` non sia una macro che rappresenta una costante
- L'espressione costante deve comunque valere un intero

L'istruzione `switch`

- In ogni "diramazione" (`case`) possiamo inserire un qualsiasi numero di istruzioni
- Non c'è bisogno delle parentesi graffe
- Normalmente l'ultima istruzione in ogni "diramazione" è l'istruzione `break`.

L'istruzione `switch`

- Non è possibile duplicare il valore di controllo delle "diramazioni"
- L'ordine dei `case` non ha importanza e la diramazione di `default` non deve essere messa necessariamente per ultima
- Più diramazioni possono "condividere" le stesse istruzioni:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

L'istruzione `switch`

- Possiamo mettere i `case` sulla stessa linea:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
            printf("Passing");
            break;
  case 0:   printf("Failing");
            break;
  default: printf("Illegal grade");
            break;
}
```

- La diramazione di `default` viene eseguita quando **nessuna** delle altre diramazioni viene eseguita (sono importanti i `break`)

Il ruolo dell'istruzione **`break`**

- L'istruzione `break` fa uscire dallo `switch`: il controllo passa all'istruzione dopo lo `switch`.
- L'istruzione `switch` può essere vista come un "salto calcolato"
- Quando l'espressione di controllo viene valutata, il controllo "salta" alla diramazione che corrisponde al valore dell'espressione di controllo.
- Quindi le etichette di un `case` servono semplicemente a denotare una posizione all'interno dello `switch`.

Il ruolo dell'istruzione `break`

- Senza `break` (o una qualche altra forma di salto) alla fine di ogni diramazione l'esecuzione continuerebbe con la prossima diramazione
- Esempio:

```
switch (grade) {
  case 4:  printf("Excellent");
  case 3:  printf("Good");
  case 2:  printf("Average");
  case 1:  printf("Poor");
  case 0:  printf("Failing");
  default: printf("Illegal grade");
}
```

- Se il valore di `grade` è 3, verrà stampato

```
GoodAveragePoorFailingIllegal grade
```

Il ruolo dell'istruzione `break`

- Un `break` può essere omesso intenzionalmente
  - Se lo si fa per errore è molto probabilmente causa di malfunzionamenti del programma
- È una buona abitudine commentare esplicitamente l'omissione di un `break`:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          num_passing++;
          /* FALL THROUGH */
  case 0: total_grades++;
          break;
}
```

- Sebbene l'ultima diramazione non richieda un `break` è buona abitudine metterlo comunque
  - Facilita l'inserimento di nuovi `case`

Programma: Stampare una data

- Contratti ed altri documenti legali necessitano spesso di una dicitura particolare per la data, del tipo:

```
Dated this _____ day of ______ , 20__ .
```

- Il programma `date.c` stampa la data in questo formato dopo avere richiesto la data all'utente nella forma usuale (mese/giorno/anno):

```
Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.
```

- Il programma usa uno `switch` per aggiungere "th" (o "st" o "nd" o "rd") al giorno, e per stampare il mese come parola e non come numero

date.c

```
/* Prints a date in legal form */

#include <stdio.h>

int main(void)
{
  int month, day, year;

  printf("Enter date (mm/dd/yy): ");
  scanf("%d/%d/%d", &month, &day, &year);

  printf("Dated this %d", day);
  switch (day) {
    case 1: case 21: case 31:
      printf("st"); break;
    case 2: case 22:
      printf("nd"); break;
    case 3: case 23:
      printf("rd"); break;
    default: printf("th"); break;
  }
  printf(" day of ");
```

```
  switch (month) {
    case 1:  printf("January");   break;
    case 2:  printf("February");  break;
    case 3:  printf("March");     break;
    case 4:  printf("April");     break;
    case 5:  printf("May");       break;
    case 6:  printf("June");      break;
    case 7:  printf("July");      break;
    case 8:  printf("August");    break;
    case 9:  printf("September"); break;
    case 10: printf("October");   break;
    case 11: printf("November");  break;
    case 12: printf("December");  break;
  }

  printf(", 20%.2d.\n", year);
  return 0;
}
```

Valori booleani nel C89

- Per molti anni nel linguaggio C non è stato previsto un vero tipo "booleano"
  - Lo standard C89 non lo prevede

- In C89 si usa una variabile `int` alla quale si assegna 0 (falso) oppure 1 (vero)

```
int flag;

flag = 0;
…
flag = 1;
```

- Funziona, anche se sarebbe meglio esplicitare il fatto che vogliamo assegnare un valore booleano

Valori booleani nel C89

- Per rendere i programmi più leggibili si possono usare due macro usando i nomi `TRUE` e `FALSE`:

```
#define TRUE 1
#define FALSE 0
```

- Assegnamenti alla variabile `flag` adesso possono essere fatti in modo più naturale e leggibile

```
flag = FALSE;
…
flag = TRUE;
```

Valori booleani nel C89

- Per controllare se `flag` è vero possiamo scrivere

  ```
  if (flag == TRUE) …
  ```

  o semplicemente

  ```
  if (flag) …
  ```

- L'ultima forma è concisa e funziona anche se la variabile `flag` assume valori diversi da 0 or 1
  - È falso solo quando vale 0
- In modo simile possiamo usare

  ```
  if (flag == FALSE) …
  ```

  oppure

  ```
  if (!flag) …
  ```

Valori booleani nel C89

- Potremmo addirittura definire una macro per specificare un "tipo" booleano

```
#define BOOL int
```

- `BOOL` può essere usato al posto di `int` nelle dichiarazioni di variabili

```
BOOL flag;
```

- Questo rende più evidente il fatto che la variabile `flag` (anche se di fatto è un `int`) contiene un valore booleano

Valori booleani nel C99

- Lo standard C99 prevede il tipo `_Bool`
- Una variabile booleana può essere dichiarata con

```
_Bool flag;
```

- `_Bool` è di fatto un `int` quindi è solo una variabile "camuffata"
- Tuttavia ad una variabile di tipo `_Bool` possiamo assegnare solo i valori 0 e 1
- Se si cerca di assegnare un qualsiasi valore diverso da 0 ad un variabile `_Bool` il valore scritto sarà sempre 1:

```
flag = 5;    /* flag is assigned 1 */
```

Valori booleani nel C99

- È permesso (ma non consigliabile) effettuare operazioni aritmetiche con variabili di tipo `_Bool`
- È permesso stampare il valore di una variabile `_Bool` (verrà stampato o 0 o 1).
- E, ovviamente, una variabile `_Bool` può essere usata come condizione di un `if`

```
if (flag)    /* tests whether flag is 1 */
  …
```

Valori booleani nel C99

- L'header file del C99 `<stdbool.h>` facilita l'uso delle variabili booleane.
- Definisce la macro, `bool`, sinonimo di `_Bool`
- Includendo `<stdbool.h>`, possiamo scrivere

```
bool flag;   /* same as _Bool flag; */
```

- `<stdbool.h>` definisce anche delle macro con i nomi `true` and `false`, che stanno per 1 e 0, permettendo quindi di scrivere

```
flag = false;
…
flag = true;
```

… arrivederci alla prossima lezione