Il linguaggio C

Istruzioni, funzioni, dati strutturati

Istruzioni

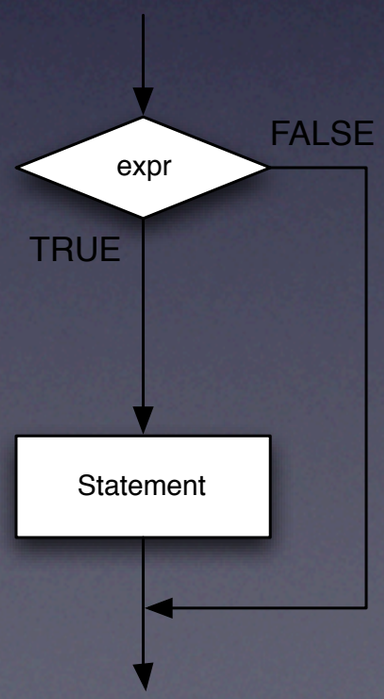
- Servono a dirigere il flusso di esecuzione di un programma
  - controllano l'ordine di esecuzione delle espressioni, quindi dei loro side effects sulle variabili
  - `<statement>::= <expr>; | ...`

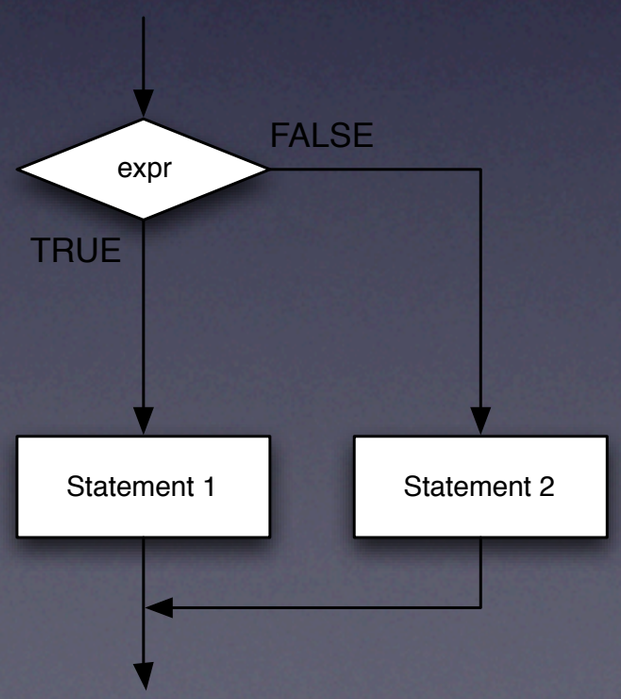
Sequenza

- <statement>::= ... |
  <statement1><statement2> |
  {<statement1>} |
  ...

Condizione

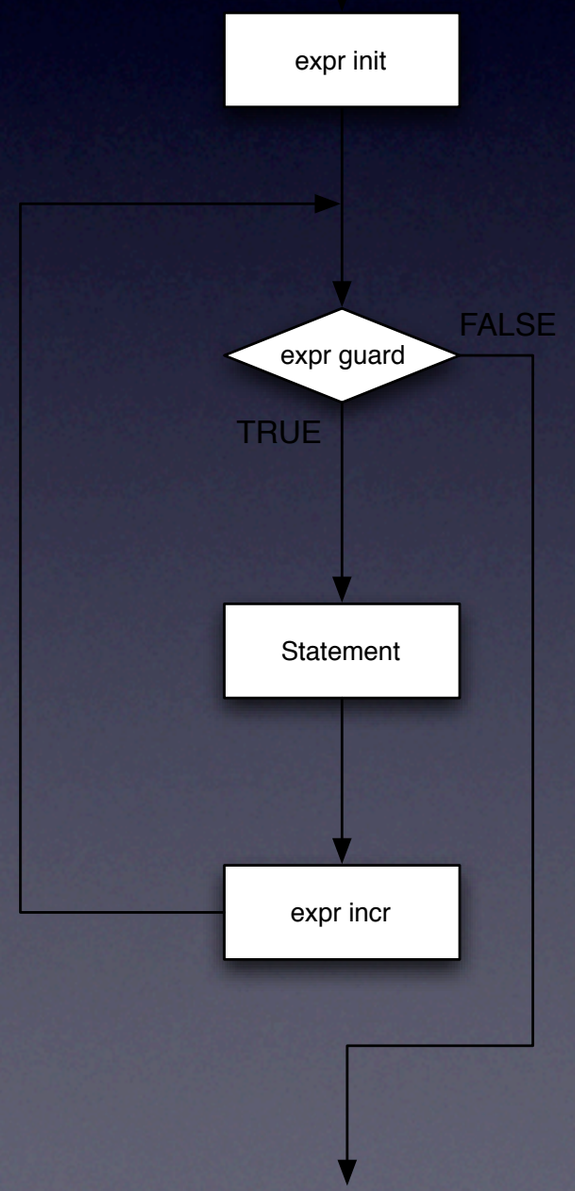
- <statement>::= ... |
  if (<expr>) <statement1>|
  if (<expr>) <statement1>
  else <statement2> |
  ...





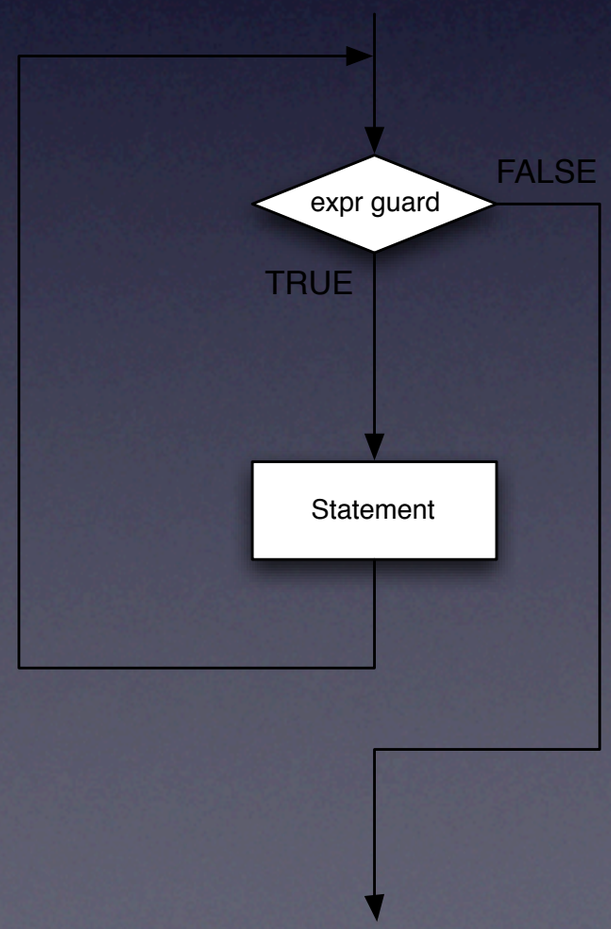
Iterazione: for

- <statement>::= ...|
for(<expr$_{init}$>;<expr$_{guard}$>;
<expr$_{inc}$>) <statement>
| ...



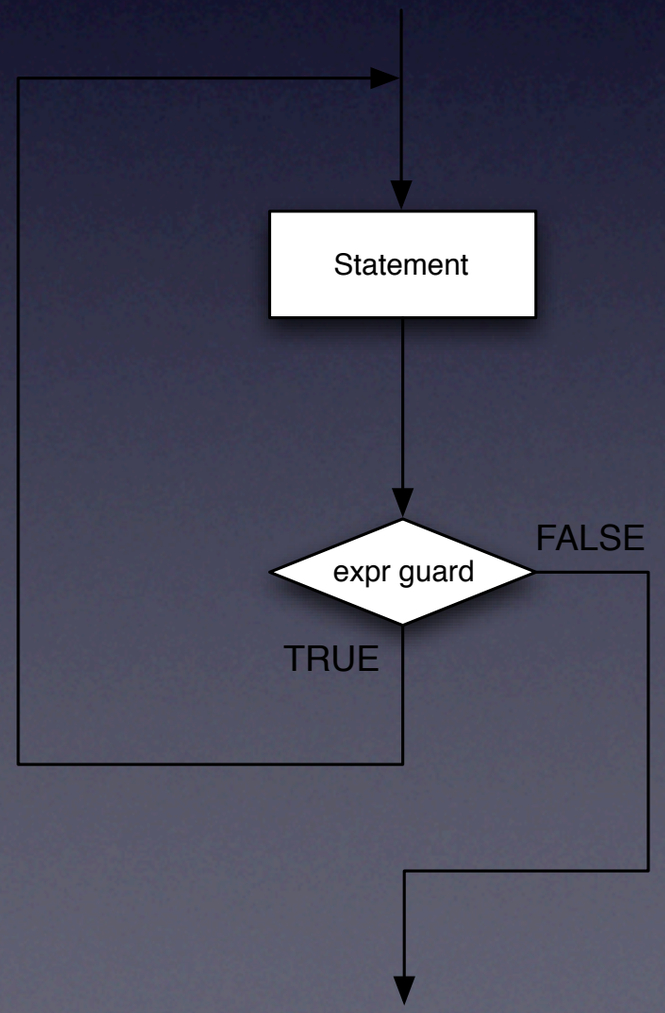
Iterazione: while

- <statement>::= ...|
  while(<expr$_{guard}$>)
  <statement> | ...



Iterazione: do-while

- <statement>::= ...| do <statement> while(<expr$_{guard}$>); | ...



Salto

- <statement>::= ... |
  goto <label>; |
  switch(<expr>)
  { {case <const>: <statement>}
  [default: <statement>]
  } |
  break; |
  continue; |

...

Funzioni

- Partizionano il programma e forniscono un meccanismo di information hiding
  - variabili locali
  - parametri formali
  - variabili globali... da usare il meno possibile

Parametri

- La risoluzione dei parametri può (in generale) essere fatta per *riferimento* o *valore*
- Per riferimento: ogni riferimento ad un parametro formale colpisce il parametro attuale: il chiamante condivide le variabili col chiamato - C++ Per valore: si creano variabili temporanee in cui vengono copiati i valori dei parametri attuali, si lavora su una copia - C e Java*

* La questione è in realtà un po' complessa. Per maggiori informazioni: http://www.javaworld.com/javaworld/javaqa/2000-05/03-qa-0526-pass.html

Parametri: uso dell'indirizzo

- Con la risoluzione per valori, per modificare il valore del parametro attuale si passa l'indirizzo della variabile, quindi con l'operatore di de-referenziazione se ne cambia il valore:

```
int param;
initParam(&param);
...
void initParam(int *pValue)
{
  *pValue=0; /* modifico il valore
                di param */
```

Parametri e variabili locali: vita

- I parametri formali sono allocati sullo stack e inizializzati con la copia del valore dei parametri formali. Le variabili locali sono allocate sullo stack
- Il tempo di vita va dall'attivazione della funzione al momento della restituzione del controllo

Parametri e variabili locali: visibilità

- La visibilità (*scope*) è la sezione di codice entro la quale si può fare riferimento alla variabile
  - si può referenziare una variabile solo nel corpo di funzione in cui è variabile locale o parametro formale
  - questo limite è "voluto" dal design del linguaggio C, per ridurre l'accoppiamento tra funzioni

- Il modificatore `static` fa rimanere in vita la variabile anche quando la funzione è terminata
- la visibilità non cambia

Variabili globali e valori restituiti

- Una variabile globale è allocata nel segmento dati del programma: ha vita coincidente con quella del programma ed è visibile da tutte le funzioni
- Una funzione può restituire un valore di un tipo
  - spesso usato per riportare l'esito della funzione

Dichiarazione di funzione

- La dichiarazione o prototipo istruisce il compilatore riguardo il nome simbolico della funzione, i parametri (nomi e tipo) ed il tipo del valore restituito

```
<declaration>::=<type><decl>;
<decl>::=...|
  <decl>(<formal param list>)
<formal param lis>::= void |
<type><decl>{,<type><decl>}
```

- Il nome della funzione denota l'indirizzo a partire dal quale è memorizzata
- simile agli array
- possiamo avere puntatori a funzione

Interpretazione

- Si estende la regola vista per array e puntatori: il modificatore suffisso `(<formal param list>)` **si interpreta come** ...*funzione che riceve in ingresso i parametri specificati in <formal param list> e restituisce...*

Esempio

- int *Y(int a, float *b);

Y è una funzione che riceve - un int ed un indirizzo di float e restituisce - un puntatore a int

Definizione di funzione

- Specifica il codice che costituisce la sezione chiamata (la dichiarazione si occupa del chiamante...)

```
<funct. definition>::=<type><decl>(<formal param list>)
{
   {<declaration>}
   <statement>
}
```

Riferimento

- La funzione è invocata con un riferimento all'indirizzo della funzione all'interno di un'espressione:
  `<expr>::= ...|`
  `<expr1>(<actual param list>)`
  `|...`
- <expr1> restituisce l'indirizzo di una funzione (es. è il nome...)
- <actual param list> è una lista di espressioni che rendono valori di tipo corrispondente a <formal param list>:
  `<actual param list>::=[<expr>{,<expr>}]`

Struttura di un programma

- <program>::= **{**
  <function definition> |
  <function declaration> |
  <directive> |
  typedef <type> <identifier>; |
  <declaration>
  }
- Una funzione non può essere definita né referenziata prima della sua dichiarazione

Tipi definiti dall'utente

- typedef definisce <identifier> come alias di <type>, es.:
  ```
  typedef unsigned char byte;
  ```
- ```
  typedef enum {a1, a2,
  a3, ...} enumType;
  ```
  il compilatore associa un numero naturale (da 0, ma si può forzare assegnando noi il valore) agli identificativi
  ```

Dati strutturati

- Insieme di variabili, anche di tipo diverso, referenziate attraverso un nome collettivo ed un indice simbolico

```
<struct_def>::= struct
<identifier>{{<var_declaration>};
```

- Si crea un nuovo tipo:

```
<type>::= ...|struct <identifier>
```

Riferimento

- Il riferimento ad una variabile strutturata segue le regole della categoria sintattica var
- Il riferimento ai campi della struttura può avvenire a partire dalla variabile o da una espressione che ne restituisce l'indirizzo

<var>::=... |
<var$_s$>.<identifier>|
<expr$_s$>-><identifier>