

Ottava dispensa: la programmazione in GNU/Linux

[(c)2002 – Jean François Panico]

Pseudocodifica

Un tempo la programmazione avveniva attraverso lunghe fasi di studio a tavolino. Prima di iniziare il lavoro di scrittura del programma (su moduli cartacei che venivano trasferiti successivamente nella macchina) si passava per la realizzazione di un diagramma di flusso, o *flow chart*. Il diagramma di flusso andava bene fino a quando si utilizzavano linguaggi di programmazione procedurali, come il COBOL. Quando si sono introdotti concetti nuovi che rendevano tale sistema di rappresentazione più complicato del linguaggio stesso, si è preferito schematizzare gli algoritmi attraverso righe di codice vero e proprio o attraverso una pseudocodifica più o meno adatta al concetto che si vuole rappresentare di volta in volta. In questo capitolo viene presentata una pseudocodifica e alcuni esempi di algoritmi tipici, utilizzabili nella didattica della programmazione. Gli esempi proposti non sono ottimizzati perché si intende puntare sulla chiarezza piuttosto che sull'eventuale velocità di esecuzione.

Descrizione

La pseudocodifica utilizzata in questo capitolo si rifà a termini e concetti comuni a molti linguaggi di programmazione recenti. Vale la pena di chiarire solo alcuni dettagli:

- le variabili di scambio di una subroutine (una procedura o una funzione) vengono semplicemente nominate a fianco del nome della procedura, tra parentesi, cosa che corrisponde a una dichiarazione implicita di quelle variabili con un campo di azione locale e con caratteristiche identiche a quelle usate nelle chiamate relative;
- il trasferimento dei parametri di una chiamata alla subroutine avviene per valore, impedendo l'alterazione delle variabili originali;
- per trasferire una variabile per riferimento, in modo che il suo valore venga aggiornato al termine dell'esecuzione di una subroutine, occorre aggiungere il simbolo @ di fronte al nome della variabile utilizzata nella chiamata;
- il simbolo # rappresenta l'inizio di un commento;
- il simbolo := rappresenta l'assegnamento;
- il simbolo ::= rappresenta lo scambio tra due operandi.

Problemi elementari di programmazione

Nelle sezioni seguenti sono descritti alcuni problemi elementari attraverso cui si insegnano le tecniche di programmazione ai principianti. Assieme ai problemi vengono proposte le soluzioni in forma di pseudocodifica.

Somma tra due numeri positivi

La somma di due numeri positivi può essere espressa attraverso il concetto dell'incremento unitario: $n+m$ equivale a incrementare m , di un'unità, per n volte, oppure incrementare n per m volte. L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli.

```
SOMMA (X, Y)
  LOCAL Z INTEGER
  LOCAL I INTEGER

  Z := X
  FOR I := 1; I <= Y; I++
    Z++
  END FOR

  RETURN Z
END SOMMA
```

In questo caso viene mostrata una soluzione per mezzo di un ciclo enumerativo, FOR. Il ciclo viene ripetuto Y volte, incrementando la variabile Z di un'unità. Alla fine, Z contiene il risultato della somma di X per Y. La pseudocodifica seguente mostra invece la traduzione del ciclo FOR in un ciclo WHILE.

```
SOMMA (X, Y)
  LOCAL Z INTEGER
  LOCAL I INTEGER

  Z := X
  I := 1
  WHILE I <= Y
    Z++
    I++
  END WHILE

  RETURN Z
END SOMMA
```

Moltiplicazione di due numeri positivi attraverso la somma

La moltiplicazione di due numeri positivi, può essere espressa attraverso il concetto della somma: $n*m$ equivale a sommare m volte n , oppure n volte m . L'algoritmo risolutivo è banale, ma utile per apprendere il funzionamento dei cicli.

```
MOLTIPLICA (X, Y)
```

```

LOCAL Z INTEGER
LOCAL I INTEGER

Z := 0
FOR I := 1; I <= Y; I++
    Z := Z + X
END FOR

RETURN Z
END MOLTIPLICA

```

In questo caso viene mostrata una soluzione per mezzo di un ciclo FOR. Il ciclo viene ripetuto Y volte, incrementando la variabile Z del valore di X. Alla fine, Z contiene il risultato del prodotto di X per Y. La pseudocodifica seguente mostra invece la traduzione del ciclo FOR in un ciclo WHILE.

```

MOLTIPLICA (X, Y)
LOCAL Z INTEGER
LOCAL I INTEGER

Z := 0
I := 1
WHILE I <= Y
    Z := Z + X
    I++
END WHILE

RETURN Z
END MOLTIPLICA

```

Divisione intera tra due numeri positivi

La divisione di due numeri positivi, può essere espressa attraverso la sottrazione: $n:m$ equivale a sottrarre m da n fino a quando n diventa inferiore di m . Il numero di volte in cui tale sottrazione ha luogo, è il risultato della divisione.

```

DIVIDI (X, Y)
LOCAL Z INTEGER
LOCAL I INTEGER

Z := 0
I := X
WHILE I >= Y
    I := I - Y
    Z++
END WHILE

RETURN Z
END DIVIDI

```

Elevamento a potenza

L'elevamento a potenza, utilizzando numeri positivi, può essere espresso attraverso il concetto della moltiplicazione: $n**m$ equivale a moltiplicare m volte n per se stesso.

```

EXP (X, Y)
LOCAL Z INTEGER
LOCAL I INTEGER

Z := 1
FOR I := 1; I <= Y; I++
    Z := Z * X
END FOR

RETURN Z
END EXP

```

In questo caso viene mostrata una soluzione per mezzo di un ciclo FOR. Il ciclo viene ripetuto Y volte; ogni volta la variabile Z viene moltiplicata per il valore di X, a partire da uno. Alla fine, Z contiene il risultato dell'elevamento di X a Y. La pseudocodifica seguente mostra invece la traduzione del ciclo FOR in un ciclo WHILE.

```

EXP (X, Y)
LOCAL Z INTEGER
LOCAL I INTEGER

Z := 1
I := 1
WHILE I <= Y
    Z := Z * X
    I++
END WHILE

```

```

    RETURN Z
END EXP

```

La pseudocodifica seguente mostra una soluzione ricorsiva.

```

EXP (X, Y)
  IF X = 0
    THEN
      RETURN 0
    ELSE
      IF Y = 0
        THEN
          RETURN 1
        ELSE
          RETURN N * EXP (N, Y-1)
        END IF
      END IF
    END IF
END EXP

```

Radice quadrata

Il calcolo della parte intera della radice quadrata di un numero si può fare per tentativi, partendo da 1, eseguendo il quadrato fino a quando il risultato è minore o uguale al valore di partenza di cui si calcola la radice.

```

RADICE (X)
  LOCAL Z INTEGER
  LOCAL T INTEGER

  Z := 0
  T := 0

  WHILE TRUE

    T := Z * Z

    IF T > X
      THEN
        # È stato superato il valore massimo.
        Z--
        RETURN Z
      END IF

    Z++

  END WHILE
END RADICE

```

Fattoriale

Il fattoriale è un valore che si calcola a partire da un numero positivo. Può essere espresso come il prodotto di n per il fattoriale di $n-1$, quando n è maggiore di 1, mentre equivale a 1 quando n è uguale a 1. In pratica, $n! = n * (n-1) * (n-2) ... * 1$.

```

FATTORIALE (X)
  LOCAL I INTEGER

  I := X - 1

  WHILE I > 0
    X := X * I
    I--
  END WHILE

  RETURN X
END FATTORIALE

```

La soluzione appena mostrata fa uso di un ciclo WHILE in cui l'indice I, che inizialmente contiene il valore di X-1, viene usato per essere moltiplicato al valore di X, riducendolo ogni volta di un'unità. Quando I raggiunge lo zero, il ciclo termina e X contiene il valore del fattoriale. L'esempio seguente mostra invece una soluzione ricorsiva che dovrebbe risultare più intuitiva.

```

FATTORIALE (X)
  IF X == 1
    THEN
      RETURN 1
    END IF

  RETURN X * FATTORIALE (X - 1)
END FATTORIALE

```

Linguaggio C: introduzione

Il linguaggio C è il fondamento dei sistemi Unix. Un minimo di conoscenza di questo linguaggio è importante per sapersi districare tra i programmi distribuiti in forma sorgente.

Il linguaggio C richiede la presenza di un compilatore per generare un file eseguibile (o interpretabile) dal kernel. Se si dispone dei cosiddetti «strumenti di sviluppo», intendendo con questo ciò che serve a ricompilare il kernel, si dovrebbe disporre di tutto quello che è necessario per provare gli esempi di questi capitoli.

Struttura fondamentale

Il contenuto di un sorgente in linguaggio C può essere suddiviso in tre parti: commenti, direttive del preprocessore e istruzioni C. I commenti vanno aperti e chiusi attraverso l'uso dei simboli /* e */.

Direttive del preprocessore

Le direttive del preprocessore rappresentano un linguaggio che guida alla compilazione del codice vero e proprio. L'uso più comune di queste direttive viene fatto per includere porzioni di codice sorgente esterne al file. È importante fare attenzione a non confondersi, dal momento che tali istruzioni iniziano con il simbolo #: non si tratta di commenti.

Il programma C tipico richiede l'inclusione di codice esterno composto da file che terminano con l'estensione .h. La libreria che viene inclusa più frequentemente è quella necessaria alla gestione dei flussi di standard input, standard output e standard error; si dichiara il suo utilizzo nel modo seguente:

```
#include <stdio.h>
```

Istruzioni C

Le istruzioni C terminano con un punto e virgola (;) e i raggruppamenti di queste si fanno utilizzando le parentesi graffe ({ }).

```
istruzione;
{istruzione; istruzione; istruzione; }
```

Generalmente, un'istruzione può essere interrotta e ripresa nella riga successiva, dal momento che la sua conclusione è dichiarata chiaramente dal punto e virgola finale. L'istruzione nulla viene rappresentata utilizzando un punto e virgola da solo.

Nomi

I nomi scelti per identificare ciò che si utilizza all'interno del programma devono seguire regole determinate, definite dal compilatore C a disposizione. Per cercare di scrivere codice portabile in altre piattaforme, conviene evitare di sfruttare caratteristiche speciali del proprio ambiente. In particolare:

- un nome può iniziare con una lettera alfabetica e continuare con altre lettere, cifre numeriche e il trattino basso;
- in teoria i nomi potrebbero iniziare anche con il trattino basso, ma questo è sconsigliabile;
- i nomi sono sensibili alla differenza tra lettere maiuscole e minuscole.

La lunghezza dei nomi può essere un elemento critico; generalmente la dimensione massima dovrebbe essere di 32 caratteri, ma ci sono versioni di C che ne possono accettare solo una quantità inferiore. In particolare, C GNU ne accetta molti di più di 32. In ogni caso, il compilatore non rifiuta i nomi troppo lunghi, semplicemente non ne distingue più la differenza oltre un certo punto.

Funzione principale

Il codice di un programma C è scomposto in funzioni, dove l'esecuzione del programma corrisponde alla chiamata della funzione main(). Questa funzione può essere dichiarata senza argomenti oppure con due argomenti precisi: main (int argc, char *argv[]).

Ciao mondo!

Come sempre, il modo migliore per introdurre a un linguaggio di programmazione è di proporre un esempio banale, ma funzionante. Al solito si tratta del programma che emette un messaggio e poi termina la sua esecuzione.

```
/*
 *      Ciao mondo!
 */

#include <stdio.h>

/* La funzione main() viene eseguita automaticamente all'avvio. */
main ()
{
    /* Si limita a emettere un messaggio. */
    printf ("Ciao mondo!\n");
}
```

Nel programma sono state inserite alcune righe di commento. In particolare, all'inizio, l'asterisco che si trova nella seconda riga non serve a nulla, se non a guidare la vista verso la conclusione del commento stesso.

Il programma si limita a emettere la stringa «Ciao Mondo!» seguita da un codice di interruzione di riga, rappresentato dal simbolo \n.

Compilazione

Per compilare un programma scritto in C si utilizza generalmente il comando cc, anche se di solito si tratta di un collegamento simbolico al vero compilatore che si ha a disposizione. Supponendo di avere salvato il file dell'esempio con il nome ciao.c, il comando per la sua compilazione è il seguente:

```
$ cc ciao.c[Invio]
```

Quello che si ottiene è il file a.out che dovrebbe già avere i permessi di esecuzione.

```
$ ./a.out[Invio]
```

```
Ciao mondo!
```

Se si desidera compilare il programma definendo un nome diverso per il codice eseguibile finale, si può utilizzare l'opzione standard -o.

```
$ cc -o ciao ciao.c[Invio]
```

Con questo comando, si ottiene l'eseguibile ciao.

```
$ ./ciao[Invio]
Ciao mondo!
```

Emissione dati attraverso printf()

L'esempio di programma presentato sopra si avvale di printf() per emettere il messaggio attraverso lo standard output. Questa funzione è più sofisticata di quanto possa apparire dall'esempio, in quanto permette di formattare il risultato da emettere. Negli esempi più semplici di codice C appare immancabilmente questa funzione, per cui è necessario descrivere subito, almeno in parte, il suo funzionamento.

```
int printf (stringa_di_formato [, espressione]...)
```

printf() emette attraverso lo standard output la stringa indicata come primo parametro, dopo averla rielaborata in base alla presenza di metavariabili riferite alle eventuali espressioni che compongono i parametri successivi. Restituisce il numero di caratteri emessi.

L'utilizzo più semplice di printf() è quello che è già stato visto, cioè l'emissione di una semplice stringa senza metavariabili (il codice \n rappresenta un carattere preciso e non è una metavariabile, piuttosto si tratta di una cosiddetta sequenza di escape).

```
printf ("Ciao mondo!\n");
```

La stringa può contenere delle metavariabili del tipo %d, %c, %f,... e queste fanno ordinatamente riferimento ai parametri successivi. Per esempio,

```
printf ("Totale fatturato: %d\n", 12345);
```

fa in modo che la stringa incorpori il valore indicato come secondo parametro, nella posizione in cui appare %d. La metavariabile %d stabilisce anche che il valore in questione deve essere trasformato secondo una rappresentazione decimale intera. Per cui, il risultato sarà esattamente quello che ci si aspetta.

```
Totale fatturato: 12345
```

Variabili e tipi

I tipi di dati elementari gestiti dal linguaggio C dipendono molto dall'architettura dell'elaboratore sottostante. In questo senso, volendo fare un discorso generale, è difficile definire la dimensione delle variabili numeriche; si può solo dare delle definizioni relative. Solitamente, il riferimento è dato dal tipo numerico intero (int) la cui dimensione in bit è data dalla dimensione della **parola**, ovvero dalla capacità dell'unità aritmetico-logica del microprocessore. In pratica, con l'architettura i386 la dimensione di un intero normale è di 32 bit.

Tipi primitivi

I tipi di dati primitivi rappresentano un valore **numerico** singolo, nel senso che anche il tipo char può essere trattato come un numero. Il loro elenco essenziale si trova nella tabella.

Tipo	Descrizione
char	Carattere (generalmente di 8 bit).
int	Intero normale.
float	Virgola mobile a singola precisione.
double	Virgola mobile a doppia precisione.

Elenco dei tipi di dati primitivi elementari in C.

Come già accennato, non si può stabilire in modo generale quali siano le dimensioni esatte in bit dei vari tipi di dati, si può solo stabilire una relazione tra loro.

```
char <= int <= float <= double
```

Questi tipi primitivi possono essere estesi attraverso l'uso di alcuni qualificatori: short, long e unsigned. I primi due si riferiscono alla dimensione, mentre l'ultimo modifica il modo di valutare il contenuto di alcune variabili. La tabella seguente riassume i vari tipi primitivi con le combinazioni dei qualificatori.

Tipo	Abbreviazione	Descrizione
char		
unsigned char		Tipo char usato numericamente senza segno.
short int	short	Intero più breve di int.
unsigned short int	unsigned short	Tipo short senza segno.
int		Intero normale.
unsigned int	unsigned	Tipo int senza segno.
long int	long	Intero più lungo di int.
unsigned long int	unsigned long	Tipo long senza segno.
float		
double		
long double		Tipo a virgola mobile più lungo di double.

Elenco dei tipi di dati primitivi in C assieme ai qualificatori.

Così, il problema di stabilire le relazioni di dimensione si complica

```
char <= short <= int <= long
float <= double <= long double
```

I tipi long e float potrebbero avere una dimensione uguale, altrimenti non è detto quale dei due sia più grande.

Il programma seguente, potrebbe essere utile per determinare la dimensione dei vari tipi primitivi nella propria piattaforma.

```

/* dimensione_variabili */

#include <stdio.h>

main ()
{
    printf ("char      %d\n", (int)sizeof(char));
    printf ("short    %d\n", (int)sizeof(short));
    printf ("int      %d\n", (int)sizeof(int));
    printf ("long     %d\n", (int)sizeof(long));
    printf ("float    %d\n", (int)sizeof(float));
    printf ("double   %d\n", (int)sizeof(double));
    printf ("long double %d\n", (int)sizeof(long double));
}

```

Il risultato potrebbe essere quello seguente:

```

char      1
short    2
int      4
long     4
float    4
double   8
long double 12

```

I numeri rappresentano la quantità di caratteri, nel senso di valori char, per cui il tipo char dovrebbe sempre avere una dimensione unitaria.

Valori contenibili

I tipi primitivi di variabili mostrati sono tutti utili alla memorizzazione di valori numerici, a vario titolo. A seconda che il valore in questione sia trattato con segno o senza segno, varia lo spettro di valori che possono essere contenuti.

Nel caso di interi (char, short, int e long), la variabile può essere utilizzata per tutta la sua estensione a contenere un numero binario. In pratica, il massimo valore ottenibile è $(2^{*n})-1$, dove n rappresenta il numero di bit a disposizione. Quando invece si vuole trattare il dato come un numero con segno, il valore numerico massimo ottenibile è circa la metà.

Nel caso di variabili a virgola mobile, non c'è più la possibilità di rappresentare esclusivamente valori senza segno; inoltre non c'è più un limite di dimensione, ma solo di approssimazione.

Le variabili char sono fatte, in linea di principio, per contenere il codice di rappresentazione di un carattere, secondo la codifica utilizzata nel sistema. Generalmente si tratta di un dato di 8 bit (1 byte), ma non è detto che debba sempre essere così. A ogni modo, il fatto che questa variabile possa essere gestita in modo numerico, permette una facile conversione da lettera a codice numerico corrispondente. Un tipo di valore che non è stato ancora visto è quello logico: *Vero* è rappresentato da un qualsiasi valore numerico diverso da zero, mentre *Falso* corrisponde a zero.

Costanti letterali

Quasi tutti i tipi di dati primitivi, hanno la possibilità di essere rappresentati in forma di costante letterale. In particolare, si distingue tra:

- costanti carattere, rappresentate da un carattere alfanumerico racchiuso tra apici singoli, come 'A', 'B',...;
- costanti intere, rappresentate da un numero senza decimali, e a seconda delle dimensioni può trattarsi di uno dei vari tipi di interi (escluso char);
- costanti con virgola, rappresentate da un numero con decimali (un punto seguito da altre cifre, anche se si tratta solo di zeri), che indipendentemente dalle dimensioni sono sempre di un tipo double.

Per esempio, 123 è generalmente una costante int, mentre 123.0 è una costante double.

Per quanto riguarda le costanti che rappresentano numeri con virgola, si può usare anche la notazione scientifica. Per esempio, 7e+15 rappresenta l'equivalente di $7 * (10^{15})$, cioè un sette con 15 zeri. Nello stesso modo, 7e-5, rappresenta l'equivalente di $7 * (10^{-5})$, cioè 0,000 07. È possibile rappresentare anche le stringhe in forma di costante attraverso l'uso degli apici doppi, ma la stringa non è un tipo di dati primitivo, trattandosi piuttosto di un array di caratteri. Per il momento è importante fare attenzione a non confondere il tipo char con la stringa. Per esempio, 'F' è un carattere, mentre "F" è una stringa, ma la differenza è notevole. Le stringhe verranno descritte meglio in seguito.

Caratteri speciali

È stato affermato che si possono rappresentare i caratteri singoli in forma di costante, utilizzando gli apici singoli come delimitatore, e che per rappresentare una stringa si usano invece gli apici doppi. Alcuni caratteri non hanno una rappresentazione grafica e non possono essere inseriti attraverso la tastiera.

In questi casi, si possono usare tre tipi di notazione: ottale, esadecimale e simbolica. In tutti i casi si utilizza la barra obliqua inversa (\) come carattere di escape, cioè come simbolo per annunciare che ciò che segue immediatamente deve essere interpretato in modo particolare. La notazione ottale usa la forma \ooo, dove ogni lettera o rappresenta una cifra ottale. A questo proposito, è opportuno notare che se la dimensione di un carattere fosse superiore ai fatidici 8 bit, occorrerebbero probabilmente più cifre (una cifra ottale rappresenta un gruppo di 3 bit).

La notazione esadecimale usa la forma \xhh, dove h rappresenta una cifra esadecimale. Anche in questo caso vale la considerazione per cui ci vorranno più di due cifre esadecimali per rappresentare un carattere più lungo di 8 bit.

La notazione simbolica permette di fare riferimento facilmente a codici di uso comune, quali <CR>, <HT>,... Inoltre, questa notazione permette anche di indicare caratteri che altrimenti verrebbero interpretati in maniera differente dal compilatore. La tabella riporta i vari tipi di rappresentazione delle costanti carattere attraverso codici di escape.

Codice di escape	Descrizione
\ooo	Notazione ottale.

Codice di escape	Descrizione
\xhh	Notazione esadecimale.
\\	Una singola barra obliqua inversa (\).
\'	Un apice singolo destro.
\"	Un apice doppio.
\0	Il codice <NUL>.
\a	Il codice <BEL> (bell).
\b	Il codice <BS> (backspace).
\f	Il codice <FF> (formfeed).
\n	Il codice <LF> (linefeed).
\r	Il codice <CR> (carriage return).
\t	Una tabulazione orizzontale (<HT>).
\v	Una tabulazione verticale (<VT>).

Elenco dei modi di rappresentazione delle costanti carattere attraverso codici di escape.

Nell'esempio introduttivo, è già stato visto l'uso della notazione \n per rappresentare l'inserzione di un codice di interruzione di riga alla fine del messaggio di saluto.

```
printf ("Ciao mondo!\n");
```

Senza di questo, il cursore resterebbe a destra del messaggio alla fine dell'esecuzione di quel programma, ponendo lì l'invito.

Campo di azione delle variabili

Il campo di azione delle variabili in C viene determinato dalla posizione in cui queste vengono dichiarate e dall'uso di particolari qualificatori. Per il momento basti tenere presente che quanto dichiarato all'interno di una funzione ha valore locale per la funzione stessa, mentre quanto dichiarato al di fuori, ha valore globale per tutto il file.

Dichiarazione delle variabili

La dichiarazione di una variabile avviene specificando il tipo e il nome della variabile, come nell'esempio seguente dove si dichiara la variabile numero di tipo intero.

```
int numero;
```

La variabile può anche essere inizializzata contestualmente, assegnandogli un valore, come nell'esempio seguente in cui viene dichiarata la stessa variabile numero con il valore iniziale di 1 000.

```
int numero = 1000;
```

Costanti simboliche

Una costante è qualcosa che non varia e generalmente si rappresenta attraverso una notazione che ne definisce il valore. Tuttavia, a volte può essere più comodo definire una costante in modo simbolico, come se fosse una variabile, per facilitarne l'utilizzo e la sua identificazione all'interno del programma. Si ottiene questo con il modificatore const. Ovviamente, è obbligatorio inizializzarla contestualmente alla sua dichiarazione. L'esempio seguente dichiara la costante simbolica pi con il valore del P-greco.

```
const float pi = 3.14159265;
```

Le costanti simboliche di questo tipo, sono delle variabili per le quali il compilatore non concede che avvengano delle modifiche.

È il caso di osservare, tuttavia, che l'uso di costanti simboliche di questo tipo è piuttosto limitato. Generalmente è preferibile utilizzare delle **macro** definite e gestite attraverso il preprocessore. L'utilizzo di queste verrà descritto più avanti.

Strutture di controllo di flusso

Il linguaggio C gestisce praticamente tutte le strutture di controllo di flusso degli altri linguaggi di programmazione, compreso *go-to* che comunque è sempre meglio non utilizzare e qui, volutamente, non viene presentato.

Le strutture di controllo permettono di sottoporre l'esecuzione di una parte di codice alla verifica di una condizione, oppure permettono di eseguire dei cicli, sempre sotto il controllo di una condizione. La parte di codice che viene sottoposta a questo controllo, può essere una singola istruzione, oppure un gruppo di istruzioni. Nel secondo caso, è necessario delimitare questo gruppo attraverso l'uso delle parentesi graffe.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre le parentesi graffe, in modo da evitare equivoci nella lettura del codice. Dato che le parentesi graffe sono usate nel codice C, se queste appaiono nei modelli sintattici indicati, queste fanno parte delle istruzioni e non della sintassi.

if

La struttura condizionale è il sistema di controllo fondamentale dell'andamento del flusso delle istruzioni.

```
if (condizione) istruzione
if (condizione) istruzione else istruzione
```

Se la condizione si verifica, viene eseguita l'istruzione o il gruppo di istruzioni che segue; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzata la sotto-struttura che si articola a partire dalla parola chiave else, nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne dipende. Sotto vengono mostrati alcuni esempi.

```
int iImporto;
...
if (iImporto > 10000000) printf ("L'offerta è vantaggiosa\n");
-----
int iImporto;
int iMemorizza;
...
if (iImporto > 10000000)
```

```

    {
        iMemorizza = iImporto;
        printf ("L'offerta è vantaggiosa\n");
    }
else
    {
        printf ("Lascia perdere\n");
    }
-----
int iImporto;
int iMemorizza;
...
if (iImporto > 10000000)
    {
        iMemorizza = iImporto;
        printf ("L'offerta è vantaggiosa\n");
    }
else if (iImporto > 5000000)
    {
        iMemorizza = iImporto;
        printf ("L'offerta è accettabile\n");
    }
else
    {
        printf ("Lascia perdere\n");
    }

```

switch

La struttura di selezione, che si attua con l'istruzione switch, è un po' troppo complessa per essere rappresentata facilmente attraverso uno schema sintattico. In generale, questa struttura permette di eseguire una o più istruzioni in base al risultato di un'espressione. L'esempio seguente mostra la visualizzazione del nome del mese, in base al valore di un intero.

```

int iMese;
...
switch (iMese)
    {
        case 1: printf ("gennaio\n"); break;
        case 2: printf ("febbraio\n"); break;
        case 3: printf ("marzo\n"); break;
        case 4: printf ("aprile\n"); break;
        case 5: printf ("maggio\n"); break;
        case 6: printf ("giugno\n"); break;
        case 7: printf ("luglio\n"); break;
        case 8: printf ("agosto\n"); break;
        case 9: printf ("settembre\n"); break;
        case 10: printf ("ottobre\n"); break;
        case 11: printf ("novembre\n"); break;
        case 12: printf ("dicembre\n"); break;
    }

```

Come si vede, dopo l'istruzione con cui si emette il nome del mese attraverso lo standard output, viene richiesta l'interruzione esplicita dell'analisi della struttura, attraverso l'istruzione break, allo scopo di togliere ambiguità al codice, garantendo che sia evitata la verifica degli altri casi. Un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso insieme di istruzioni.

```

int iAnno;
int iMese;
int iGiorni;
...
switch (iMese)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            iGiorni = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            iGiorni = 30;
            break;
        case 2:

```

```

    if (((iAnno % 4 == 0) && !(iAnno % 100 = 0)) ||
        (iAnno % 400 == 0))
        iGiorni = 29;
    else
        iGiorni = 28;
    break;
}

```

È anche possibile definire un caso predefinito che si verifica quando nessuno degli altri si avvera.

```

int iMese;
...
switch (iMese)
{
    case 1: printf ("gennaio\n"); break;
    case 2: printf ("febbraio\n"); break;
    ...
    case 11: printf ("novembre\n"); break;
    case 12: printf ("dicembre\n"); break;
    default: printf ("mese non corretto\n"); break;
}

```

while

while (condizione) istruzione

L'iterazione si ottiene normalmente in C attraverso l'istruzione while, che esegue un'istruzione, o un gruppo di queste, finché la condizione continua a restituire il valore *Vero*. La condizione viene valutata prima di eseguire il gruppo di istruzioni e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo. L'esempio seguente fa apparire per 10 volte la lettera «x».

```

int iContatore = 0;
while (iContatore < 10)
{
    iContatore++;
    printf ("x");
}
printf ("\n");

```

Nel blocco di istruzioni di un ciclo while, ne possono apparire alcune particolari:

- break, che serve a uscire definitivamente dalla struttura del ciclo;
- continue, che serve a interrompere l'esecuzione del gruppo di istruzioni, riprendendo immediatamente con il ciclo successivo (a partire dalla valutazione della condizione).

L'esempio seguente è una variante del calcolo di visualizzazione mostrato sopra, modificato in modo da vedere il funzionamento dell'istruzione break. All'inizio della struttura, while (1) equivale a stabilire che il ciclo è senza fine, perché la condizione è sempre vera. In questo modo, solo la richiesta esplicita di interruzione dell'esecuzione della struttura (attraverso l'istruzione break) permette l'uscita da questa.

```

int iContatore = 0;

while (1)
{
    if (iContatore >= 10)
    {
        break;
    }
    iContatore++;
    printf ("x");
}
printf ("\n");

```

do-while

Una variante del ciclo while, in cui l'analisi della condizione di uscita avviene dopo l'esecuzione del blocco di istruzioni che viene iterato, è definito dall'istruzione do.

do blocco_di_istruzioni while (condizione);

In questo caso, si esegue un gruppo di istruzioni una volta, poi se ne ripete l'esecuzione finché la condizione restituisce il valore *Vero*.

for

In presenza di iterazioni in cui si deve incrementare o decrementare una variabile a ogni ciclo, si usa preferibilmente la struttura for, che in C permetterebbe un utilizzo più ampio di quello comune.

for (espressione1; espressione2; espressione3) istruzione

Questa è la forma tipica di un'istruzione for, in cui la prima espressione corrisponde all'assegnamento iniziale di una variabile, la seconda a una condizione che deve verificarsi fino a che si vuole che sia eseguita l'istruzione (o il gruppo di istruzioni) e la terza all'incremento o decremento della variabile inizializzata con la prima espressione. In pratica, potrebbe esprimersi nella sintassi seguente:

for (var = n; condizione; var++) istruzione

Il ciclo for potrebbe essere definito anche in maniera differente, più generale: la prima espressione viene eseguita una volta sola all'inizio del ciclo; la seconda viene valutata all'inizio di ogni ciclo e il gruppo di istruzioni viene eseguito solo se il risultato è *Vero*; l'ultima viene eseguita alla fine dell'esecuzione del gruppo di istruzioni, prima che si ricominci con l'analisi della condizione.

L'esempio già visto, in cui veniva visualizzata per dieci volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo for.

```
int iContatore;

for (iContatore = 0; iContatore < 10; iContatore++)
{
    printf ("x");
}
printf ("\n");
```

Anche nelle istruzioni controllate da un ciclo for si possono collocare istruzioni break e continue, con lo stesso significato visto per il ciclo while. Sfruttando la possibilità di inserire più espressioni in una singola istruzione, si possono realizzare dei cicli for molto più complessi, anche se però questo è sconsigliabile per evitare di scrivere codice troppo difficile da interpretare. In questo modo, l'esempio precedente potrebbe essere ridotto a quello che segue:

```
int iContatore;

for (iContatore = 0; iContatore < 10; printf ("x"), iContatore++)
{
    ;
}
printf ("\n");
```

Il punto e virgola solitario rappresenta un'istruzione nulla.

Funzioni

Il linguaggio C offre le funzioni come mezzo per realizzare la scomposizione del codice in subroutine. Prima di poter essere utilizzate attraverso una chiamata, le funzioni devono essere dichiarate, anche se non necessariamente descritte. In pratica, se si vuole indicare nel codice una chiamata a una funzione che viene descritta più avanti, occorre almeno dichiararne il prototipo.

Le funzioni del linguaggio C prevedono il passaggio di parametri solo **per valore**, con tipi di dati primitivi (compresi i puntatori che verranno descritti nel prossimo capitolo). Il linguaggio C offre un gran numero di funzioni interne, che vengono importate nel codice attraverso l'istruzione #include del preprocessore. In pratica, in questo modo si importa la parte di codice necessaria alla dichiarazione e descrizione di queste funzioni standard. Per esempio, come si è già visto, per poter utilizzare la funzione printf() si deve inserire la riga #include <stdio.h> nella parte iniziale del file sorgente.

Dichiarazione di un prototipo

tipo nome ([tipo_parametro[,...]]);

Quando la descrizione di una funzione può essere fatta solo dopo l'apparizione di una sua chiamata, occorre dichiararne il prototipo all'inizio, secondo la sintassi appena mostrata. Il tipo, posto all'inizio, rappresenta il tipo di valore che la funzione restituisce. Se la funzione non deve restituire alcunché, si utilizza il tipo void. Se la funzione richiede dei parametri, il tipo di questi deve essere elencato tra le parentesi tonde. L'istruzione con cui si dichiara il prototipo termina regolarmente con un punto e virgola.

Lo standard C ANSI stabilisce che una funzione che non richiede parametri deve utilizzare l'identificatore void in modo esplicito, all'interno delle parentesi.

Esempi

```
int fattoriale (int);
```

In questo caso, viene dichiarato il prototipo della funzione fattoriale, che richiede un parametro di tipo int e restituisce anche un valore di tipo int.

```
void elenca ();
```

Si tratta della dichiarazione di una funzione che fa qualcosa senza bisogno di ricevere alcun parametro e senza restituire alcun valore (void).

```
void elenca (void);
```

Esattamente come nell'esempio precedente, solo che è indicato in modo esplicito il fatto che la funzione non riceve argomenti (il tipo void è stato messo all'interno delle parentesi), come richiede lo standard ANSI.

Descrizione di una funzione (prototipo)

La descrizione della funzione, rispetto alla dichiarazione del prototipo, aggiunge l'indicazione dei nomi da usare per identificare i parametri e naturalmente le istruzioni da eseguire. Le parentesi graffe che appaiono nello schema sintattico fanno parte delle istruzioni necessarie.

tipo nome ([tipo parametro[,...]]) {istruzione;... }

Per esempio, la funzione seguente esegue il prodotto tra i due parametri forniti e ne restituisce il risultato.

```
int prodotto (int x, int y)
{
    return x*y;
}
```

I parametri indicati tra parentesi, rappresentano una dichiarazione di variabili locali che conterranno inizialmente i valori usati nella chiamata. Il valore restituito dalla funzione viene definito attraverso l'istruzione return, come si può osservare dall'esempio. Naturalmente, le funzioni di tipo void, cioè quelle che non devono restituire alcun valore, non hanno questa istruzione.

Variabili locali e globali

Le variabili dichiarate all'interno di una funzione, oltre a quelle dichiarate implicitamente come mezzo di trasporto dei parametri, sono visibili solo al suo interno, mentre quelle dichiarate al di fuori, dette globali, sono accessibili a tutte le funzioni. Se una variabile locale ha un nome coincidente con quello di una variabile globale, allora, all'interno della funzione, quella variabile globale non sarà accessibile.

Le regole da seguire per scrivere programmi chiari e facilmente modificabili, prevedono che si debba fare in modo di rendere le funzioni indipendenti dalle variabili globali, fornendo loro tutte le informazioni necessarie attraverso i parametri della chiamata. In questo modo diventa del tutto indifferente il fatto che una variabile locale vada a mascherare una variabile globale; inoltre, ciò permette di non dover tenere a mente il ruolo di queste variabili globali. In pratica, ci sono situazioni in cui può avere senso l'utilizzo di variabili globali per fornire informazioni alle funzioni, tuttavia occorre giudizio, come in ogni cosa.

Struttura e campo di azione

Un programma scritto in linguaggio C può essere articolato in diversi file sorgenti, all'interno dei quali si può fare riferimento solo a «oggetti» dichiarati preventivamente. Questi oggetti sono variabili e funzioni: la loro dichiarazione non corrisponde necessariamente con la loro descrizione che può essere collocata altrove, nello stesso file o in un altro file sorgente del programma.

Funzioni

Quando si vuole fare riferimento a una funzione descritta in un file sorgente differente, o in una posizione successiva dello stesso file, occorre dichiararne il prototipo in una posizione precedente. Se si desidera fare in modo che una funzione sia accessibile solo nel file sorgente in cui viene descritta, occorre definirla come static.

```
static void miafunzione (...)  
{  
...  
}
```

Variabili e classi di memorizzazione

Quando si dichiarano delle variabili, senza specificare alcuna classe di memorizzazione (cioè quando lo si fa normalmente come negli esempi visti fino a questo punto), il loro campo di azione è relativo alla posizione della dichiarazione:

- le variabili dichiarate all'esterno delle funzioni sono globali, cioè accessibili da parte di tutte le funzioni, a partire dal punto in cui vengono dichiarate;
- le variabili dichiarate all'interno delle funzioni sono locali, cioè accessibili esclusivamente dall'interno della funzione in cui si trovano.

Si distinguono quattro tipi di classi di memorizzazione, a cui corrisponde una parola chiave per la loro dichiarazione:

- automatica, auto;
- registro, register;
- statica, static;
- esterna, extern.

La prima, auto, è la classe normale: vale in modo predefinito e non occorre indicarla quando si dichiarano le variabili (variabili automatiche). Una variabile dichiarata come appartenente alla classe register, viene posta in un registro del microprocessore. Ciò può essere utile per velocizzare l'esecuzione di un programma che deve accedere frequentemente a una certa variabile, ma generalmente l'utilizzo di questa tecnica è sconsigliabile.

La classe di memorizzazione static genera due situazioni distinte, a seconda della posizione in cui viene dichiarata la variabile. Se si tratta di una variabile globale, cioè definita al di fuori delle funzioni, risulterà accessibile solo all'interno del file sorgente in cui viene descritta. Se invece si tratta di una variabile locale, cioè interna a una funzione, si tratta di una variabile che mantiene il suo valore tra una chiamata e l'altra. In questo senso, una variabile locale statica, richiede generalmente un'inizializzazione all'atto della dichiarazione; tale inizializzazione avverrà una sola volta, all'avvio del programma.

Quando da un file sorgente si vuole accedere a variabili globali dichiarate in modo normale in un altro file, oppure, quando nello stesso file si vuole poter accedere a variabili dichiarate in una posizione più avanzata dello stesso, occorre una sorta di prototipo delle variabili: la dichiarazione extern. In questo modo si informa esplicitamente il compilatore e il linker della presenza di queste.

Esempi

```
int accumula (int iAggiunta)  
{  
    static int iAccumulo = 0;  
    iAccumulo += iAggiunta;  
    return iAccumulo;  
}
```

La funzione appena mostrata si occupa di accumulare un valore e di restituirne il livello raggiunto a ogni chiamata. Come si può osservare, la variabile statica iAccumulo viene inizializzata a zero, altrimenti non ci sarebbe modo di cominciare con un valore di partenza corretto.

```
-----  
static int iMiaVariabile;  
...  
int miafunzione (...)  
{  
...  
}
```

La variabile `iMiaVariabile` è accessibile solo alle funzioni descritte nello stesso file in cui si trova, impedendo l'accesso a questa da parte di funzioni di altri file attraverso la dichiarazione `extern`.

```
-----  
extern int iMiaVariabile;  
...  
int miafunzione (...)  
{  
    iMiaVariabile = ...  
}  
int iMiaVariabile = 123;  
...
```

In questo esempio, la variabile `iMiaVariabile` è dichiarata formalmente in una posizione centrale del file sorgente; per fare in modo che la funzione `miafunzione` possa accedervi, è stata necessaria la dichiarazione `extern` iniziale.

```
-----  
extern int iTuaVariabile;  
...  
int miafunzione (...)  
{  
    iTuaVariabile = ...  
}  
...
```

Questo caso rappresenta la situazione in cui una variabile dichiarata in un altro file sorgente diventa accessibile alle funzioni del file attuale attraverso la dichiarazione `extern`. Perché ciò possa funzionare, occorre che la variabile `iTuaVariabile` sia stata dichiarata in modo normale, senza la parola chiave `static`.