Parte xxvii

Pascal

«

Pascal: preparazione di alcuni compilatori comuni

«

Sono disponibili diversi compilatori per il linguaggio Pascal; in questo capitolo ne vengono descritti alcuni.

Pascal-to-C

Pascal-to-C,[1] è una sorta di compilatore che permette di convertire un sorgente Pascal in un sorgente C. I problemi che possono sorgere da questo tipo di conversione sono nella definizione precisa del tipo di dialetto Pascal e del tipo di dialetto C. Utilizzando Pascal-to-C con GNU/Linux, non si dovrebbero avere difficoltà con il compilatore C. Quello che resta da sistemare è la definizione del dialetto Pascal che si vuole usare, dal momento che ne esistono di diversi, che alle volte sono incompatibili.

Questi dettagli possono essere controllati e configurati; quello che conta è esserne consapevoli e approfondire l'uso di Pascal-to-C attraverso lo studio della documentazione originale, quando se ne presenta la necessità, ovvero quando si intende programmare seriamente attraverso questo strumento.

«

Il nome di Pascal-to-C è indicato dal suo autore come P2c. Tuttavia, P2C è anche il nome di un altro compilatore analogo, realizzato per sistemi speciali: *http://www.geocities.com/SiliconValley/Network/3656/rocket/* . In questo secondo caso, oltre alla particolarità del compilatore stesso, c'è da considerare il fatto che non si tratta di software libero.

Librerie e compilazione

«

Il codice C generato da Pascal-to-C contiene sempre l'inclusione del file '`p2c/p2c.h`', che poi, a sua volta, provvede a includere il solito '`stdio.h`'.

Il *link* del file generato dalla compilazione del sorgente C che si ottiene, deve essere fatto includendo la libreria '`libp2c.a`', cosa che si traduce generalmente nell'uso dell'opzione '**`-lp2c`**'.

In pratica, le fasi necessarie a ottenere un programma eseguibile si riassumono nei due comandi seguenti.

```
p2c sorgente_pascal
```

```
cc -lp2c sorgente_c
```

L'eseguibile che si ottiene, richiede la presenza della libreria dinamica '`libp2c.so`'.

Il funzionamento predefinito di '**`p2c`**' può essere configurato attraverso una serie di file di configurazione:

1. '`/usr/lib/p2c/p2crc`', '**`$P2CRC`**'

2. ‘`~/p2crc`’
3. ‘`~/.p2crc`’

Il primo file dell’elenco è quello usato per definire la configurazione generale. Eventualmente, si può usare la variabile di ambiente ‘**`P2CRC`**’, contenente il percorso assoluto per raggiungere un file analogo, sostituendosi in tal modo a quello generale.

Dopo il file di configurazione generale, viene cercato il file ‘`p2crc`’ nella directory personale dell’utente, oppure, in sua mancanza, il file ‘`.p2crc`’. Questo file serve a definire una personalizzazione della configurazione di ‘**`p2c`**’.

Le direttive di questo file di configurazione sono rappresentate da assegnamenti, espressi in una delle due forme seguenti.

```
nome = valore
```

```
nome valore
```

I commenti si rappresentano come di consueto facendoli precedere dal simbolo ‘**`#`**’, dove le righe vuote o bianche vengono semplicemente ignorate.

Il file di configurazione che accompagna Pascal-to-C, cioè ‘`/usr/lib/p2c/p2crc`’, contiene l’elenco completo di tutte le direttive utilizzabili, tutte impostate nel modo più conveniente per l’uso normale e tutte debitamente commentate in modo da sapere come può essere modificato ogni valore.

L’esempio seguente definisce l’utilizzo di un sorgente TURBO Pascal:

```
Language Turbo
```

Le direttive di configurazione possono anche essere incorporate all’interno dello stesso sorgente Pascal, permettendo così una definizione dinamica, riferita a porzioni di codice. Per farlo, si utilizza una forma speciale dei commenti Pascal (le parentesi graffe fanno parte della direttiva).

{ ***nome*=*valore*** }

In tal caso, come si può vedere, il simbolo ‘**=**’ è obbligatorio e l’uso di spazi bianchi è generalmente inammissibile. È possibile l’utilizzo di commenti anche all’interno di direttive espresse in questo modo. Per farlo, occorre usare la sequenza ‘**##**’.

La configurazione dinamica all’interno del sorgente, permette di utilizzare anche altre modalità di assegnamento e di eliminazione automatica delle definizioni alla fine del sorgente. Per approfondirle, conviene consultare la documentazione originale, cosa che si riduce in pratica alla lettura di *p2c(1)*.

Segue la descrizione di alcuni esempi.

- `{Language=Turbo}`

  Definisce l’utilizzo di un sorgente TURBO Pascal.

- `{Language=Turbo ## utilizza una codifica TURBO Pascal}`

  Definisce l’utilizzo di un sorgente TURBO Pascal e vi aggiunge un commento interno.

Le direttive della configurazione di Pascal-to-C sono numerose; anche se l'impostazione predefinita si adatta alle situazioni più comuni, potrebbe essere conveniente modificarne alcune, già le prime volte che si utilizza Pascal-to-C.

```
AnsiC [0|1]
```

Permette di definire il tipo di dialetto C da utilizzare. Se si attiva la modalità, utilizzando il valore uno, si fa in modo di generare codice C ANSI; se invece non si inserisce, o si utilizza il valore zero, si ottiene un codice compatibile con il C K&R originale.

Come accennato, se non si definisce diversamente, si ottiene un codice C tradizionale, mentre potrebbe essere desiderabile di generare codice C ANSI.

```
Language [HP|HP-UX|Turbo|UCSD|VAX|Oregon|Berk|Modula]
```

Permette di definire il dialetto Pascal utilizzato come sorgente per la conversione. Le varie parole chiave usate per distinguere i dialetti hanno il valore seguente:

'**HP**'	È la codifica usata in modo predefinito e si riferisce precisamente al Pascal HP, compatibile con il Pascal dello standard ISO.
'**HP-UX**'	È il Pascal HP del sistema HP-UX, praticamente identico al Pascal HP normale.
'**Turbo**'	TURBO Pascal 5.0, quello usato con il Dos. La differenza rispetto al tipo HP è minima, tanto che generalmente non è necessario richiedere esplicitamente questo tipo di codifica, quando si usano sorgenti TURBO Pascal.

'**UCSD**'	UCSD Pascal. Si tratta di un dialetto molto simile al TURBO Pascal.
'**MPW**'	Macintosh Programmer's Workshop Pascal 2.0, senza le estensioni Object Pascal.
'**VAX**'	VAX/VMS Pascal 3.5. Non tutte le funzionalità sono disponibili.
'**Oregon**'	Oregon Software Pascal/2.
'**Berk**'	Berkeley Pascal con le estensioni Sun.
'**Modula**'	Modula-2. Basato sul libro *Programming in Modula-2* di Wirth, terza edizione. La conversione in C a partire da questo formato non è ancora completa.

```
ShortOpt [0|1]
```

Permette di definire il modo con cui devono essere valutate le espressioni logiche: uno abilita il «cortocircuito» attraverso cui si valutano effettivamente solo le condizioni strettamente necessarie a determinare il risultato finale; zero lo disabilita, in modo che tutte le condizioni vengano valutate in ogni caso.

Uso di Pascal-to-C

«

La conversione del sorgente Pascal in linguaggio C avviene per mezzo del programma '**p2c**', configurato come descritto nelle sezioni precedenti.

'**p2c**' è effettivamente un compilatore, il cui risultato è un programma C. Questo significa che genera da solo la segnalazione di errori di sintassi nel sorgente Pascal e, alla fine, il sorgente C che si ottiene dovrebbe essere corretto (dal punto di vista del C).

```
p2c [opzioni] [file]
```

‘**p2c**’ legge il file indicato come argomento, oppure lo standard input in sua mancanza. In base alle opzioni e alla configurazione definita, genera da quel file una trasformazione in linguaggio C.

Il nome del file generato si ottiene togliendo l’eventuale estensione precedente e aggiungendo ‘`.c`’.

Tabella u118.5. Alcune opzioni di ‘**p2c**’.

Opzione	Descrizione
`-o` ***file***	Definisce esplicitamente il nome del file del sorgente C da generare.
`-c` ***file_di_configurazione***	Definisce il nome di un file di configurazione da utilizzare al posto di quelli standard.
`-a`	Genera codice C ANSI. Questa opzione permette di sostituirsi agevolmente alla configurazione standard secondo cui il sorgente generato dovrebbe essere di tipo tradizionale (K&R).
`-l {HP|HP-UX|Turbo|UCSD|VAX|Oregon|Berk|Modula}`	Permette di definire il tipo di Pascal nel sorgente. Le caratteristiche abbinate alle varie parole chiave sono state descritte in occasione della descrizione dei file di configurazione.

Segue la descrizione di alcuni esempi.

- `$` **`p2c mio_programma.pas`** [*Invio*]

  Genera il file ‘`mio_programma.c`’ convertendo il contenuto di

‘mio_programma.pas’.

- $ **p2c -a mio_programma.pas** [*Invio*]

  Come nell’esempio precedente, ma genere un programma C secondo lo standard ANSI.

- $ **p2c -a -o mio.c mio_programma.pas** [*Invio*]

  Come nell’esempio precedente, ma il file generato è ‘mio.c’.

Segue la descrizione di un esempio di compilazione:

```
{
    CiaoMondo.pas
    Programma elementare di visualizzazione di un messaggio
    attraverso lo standard output.
}

program CiaoMondo;

begin
    Writeln('Ciao Mondo!');
end.
```

Se il file si chiama ‘CiaoMondo.pas’, si può trasformare in C con il comando seguente:

$ **p2c CiaoMondo.pas** [*Invio*]

```
CiaoMondo

Translation completed
```

Si ottiene così il file ‘CiaoMondo.c’, mostrato di seguito.

```
/* Output from p2c, the Pascal-to-C translator */
/* From input file "CiaoMondo.pas" */



/*
    CiaoMondo.pas
    Programma elementare di visualizzazione di un messaggio
```

```
    attraverso lo standard output.
*/


#include <p2c/p2c.h>


main(argc, argv)
int argc;
Char *argv[];
{
  PASCAL_MAIN(argc, argv);
  printf("Ciao Mondo!\n");
  exit(EXIT_SUCCESS);
}



/* End. */
```

Questo file può essere compilato a sua volta.

```
$ cc -lp2c -o CiaoMondo CiaoMondo.c [Invio]
```

Se tutto funziona correttamente, si ottiene il file 'CiaoMondo' eseguibile.

```
$ ./CiaoMondo [Invio]
```

```
Ciao Mondo!
```

Se si desidera generare un sorgente C ANSI, si può usare l'opzione '**-a**' di '**p2c**'. Nel caso dell'esempio, il corpo del programma C sarebbe stato il seguente:

```
main(int argc, Char *argv[])
{
  PASCAL_MAIN(argc, argv);
  printf("Ciao Mondo!\n");
  exit(EXIT_SUCCESS);
}
```

GPC

«

GPC, [2] ovvero il Pascal GNU, è un compilatore Pascal che fa parte di GCC. Il suo utilizzo immediato è molto semplice:

```
gpc file_pascal
```

In questo modo, l'eseguibile '**gpc**' compila il sorgente indicato come argomento e genera il file 'a.out', che poi può essere avviato. In alternativa, per specificare il nome del file eseguibile da generare con la compilazione, si usa l'opzione '**-o**':

```
gpc -o file_eseguibile file_pascal
```

Per esempio, per compilare il programma seguente, contenuto nel file 'CiaoMondo.pas', si può procedere con il comando mostrato subito dopo:

```
{
    CiaoMondo.pas
    Programma elementare di visualizzazione di un messaggio
    attraverso lo standard output.
}

program CiaoMondo;

begin
    Writeln('Ciao Mondo!');
end.
```

```
$ gpc -o CiaoMondo CiaoMondo.pas [Invio]
```

Dalla compilazione si genera il file eseguibile 'CiaoMondo':

```
$ ./CiaoMondo [Invio]
```

GPC è un compilatore Pascal molto sofisticato, accompagnato da una documentazione molto dettagliata. Qui si elencano soltanto le opzioni di uso più comune, per consentirne l'uso a scopo didattico, con gli esempi che vengono descritti nei capitoli successivi di questa parte dedicata al Pascal.

Tabella u118.13. Alcune opzioni per l'uso del compilatore GPC.

Opzione	Descrizione
`-o` ***file***	Specifica il nome del file eseguibile o del file oggetto da generare.
`-Wall`	Richiede di mostrare tutte le informazioni diagnostiche (*warning*) che possono essere utili a migliorare il sorgente Pascal.
`-Werror`	Fa in modo che ogni piccolo difetto del sorgente sia considerato alla stregua di un errore che impedisce il buon esito della compilazione.
`-static`	Se possibile, fa in modo di incorporare le librerie nell'eseguibile che viene generato.
`--short-cirquit` `--no-short-cirquit`	Abilita o disabilita il «cortocircuito», ovvero la valutazione effettiva delle sole condizioni strettamente necessarie a determinare il risultato finale.

Free-Pascal

Free-Pascal [3] è un altro compilatore Pascal. La compilazione è svolta dall'eseguibile '**`fpc`**' che si avvale anche di un file di configurazione, '`/etc/fpc.cfg`', che può contenere le stesse opzioni della riga di comando.

In condizioni normali, quando si installa Free-Pascal da un pacchetto già pronto per la propria distribuzione GNU, il file di configurazione dovrebbe essere già adatto alle caratteristiche del proprio sistema, senza richiedere altri interventi. Così, di solito è sufficiente compilare un programma in questo modo:

```
fpc file_pascal
```

Di solito, se il nome del file sorgente Pascal ha un'estensione del tipo '`.pas`', si ottiene un file eseguibile con la stessa radice e senza estensione. Per esempio, compilando il file seguente, denominato '`CiaoMondo.pas`', si ottiene il file eseguibile '`ciaomondo`' nella stessa directory:

```
{
    CiaoMondo.pas
    Programma elementare di visualizzazione di un messaggio
    attraverso lo standard output.
}

program CiaoMondo;

begin
    Writeln('Ciao Mondo!');
end.
```

```
$ fpc CiaoMondo.pas [Invio]
```

Si osservi che il nome dell'eseguibile che si ottiene contiene solo lettere minuscole:

```
$ ./ciaomondo [Invio]
```

```
Ciao Mondo!
```

[1] **Pascal-to-C** GNU GPL

[2] **GPC** GNU GPL

[3] **Free-Pascal** GNU LGPL

Pascal: introduzione

Il linguaggio Pascal è nato come strumento puramente didattico, che poi si è esteso fino a raggiungere potenzialità vicine a quelle del linguaggio C.

La caratteristica più appariscente di questo linguaggio è che tutto deve essere dichiarato prima del suo utilizzo. Il vantaggio di questo tipo di approccio sta nella possibilità di escludere errori di programmazione dovuti a digitazione errata dei nomi delle variabili, perché il compilatore si rifiuta di considerarle se non sono state dichiarate preventivamente.

Dal momento che di dialetti Pascal ne esistono molti, in questo capitolo si cerca di fare riferimento allo standard ANSI, anche se potrebbe essere particolarmente riduttivo. Gli esempi che vengono proposti dovrebbero essere compatibili con i compilatori descritti nel capitolo u118, senza bisogno di configurazioni particolari.

Struttura fondamentale

Il Pascal impone una struttura nella preparazione dei sorgenti. L'esempio seguente è un programma che non fa alcunché.

```
program Nulla;

begin
end.
```

Nella prima riga dell'esempio, si può osservare la definizione del nome del programma, attraverso la direttiva '**program**'. Il nome, in questo caso è '**Nulla**', non deve corrispondere necessariamente al nome del file.

Le parole chiave '**begin**' e '**end**' delimitano lo spazio utilizzato per le istruzioni del programma, che in questo caso non esistono.

Il punto finale, dopo la parola chiave '**end**', serve a indicare al compilatore la conclusione del programma, che può apparire solo alla fine del sorgente.

Istruzioni Pascal

Le istruzioni Pascal terminano con un punto e virgola ('**;**'), così un'istruzione può impiegare più righe senza bisogno di utilizzare simboli di continuazione, oppure, su una riga possono apparire più istruzioni (sempre separate con il punto e virgola).

È possibile raggruppare più istruzioni attraverso i delimitatori '**begin**' e '**end**': il primo dei due viene seguito dalle istruzioni senza l'uso del punto e virgola, mentre il secondo termina normalmente con un punto e virgola, oppure un punto se si tratta del delimitatore che conclude il programma.

```
istruzione ;
```

```
begin istruzione ; istruzione ; istruzione ; end;
```

L'istruzione nulla può essere rappresentata da un punto e virgola isolato.

Nomi

«

Secondo il Pascal standard, i nomi che servono per identificare ciò che si utilizza, come variabili, procedure o funzioni, sono composti da una lettera alfabetica, seguita da una combinazione libera di altre lettere e cifre numeriche. Secondo lo standard originale non è ammissibile l'uso del trattino basso, ma la maggior parte dei compilatori ammette anche questo carattere.

La lunghezza dei nomi dovrebbe essere libera, con la limitazione che ogni compilatore è in grado di distinguere i nomi solo in base a un numero massimo di caratteri. Il valore minimo definito dallo standard è di otto caratteri.

Per quanto riguarda i nomi, il Pascal non distingue tra maiuscole e minuscole, come invece avviene nel linguaggio C.

Commenti

«

Il Pascal consente l'utilizzo di due tipi di delimitatore per circoscrivere i commenti: le parentesi graffe ('`{`' e '`}`') e la coppia '`(*`' '`*)`'. Generalmente non sono ammissibili i commenti annidati, cioè quelli a più livelli.

Quello che segue è l'esempio del programma che non fa alcunché, con qualche commento.

```
{
    Ecco un programma che non fa proprio nulla.
}
program Nulla;

begin
    (* è qui che ha luogo il «nulla» *)
end.
```

Esistono due tipi di delimitatori per i commenti solo perché i primi, cioè le parentesi graffe, sono difficili da ottenere nelle prime tastiere di alcuni paesi europei.

Suddivisione di un programma Pascal

Il linguaggio Pascal è un po' rigido per ciò che riguarda la sequenza con cui possono essere descritte le varie parti che lo compongono. Si distinguono tre parti fondamentali nel file sorgente:

1. intestazione del programma -- si tratta della dichiarazione '**`program`**' seguita dal nome;
2. dichiarazioni -- è lo spazio in cui si dichiara tutto ciò che viene usato nel programma, per esempio le variabili, le procedure e le funzioni;
3. istruzioni -- è lo spazio, delimitato dalle parole chiave '**`begin`**' '**`end`**', in cui si inseriscono le istruzioni del programma, ovvero è quello che in altri linguaggi di programmazione è la funzione o la procedura principale.

È il caso di osservare che i commenti possono essere collocati in ogni punto del file sorgente.

Output elementare

«

Quasi tutti gli esempi di programmazione elementare, in qualunque linguaggio di programmazione, utilizzano un'istruzione per l'output elementare.

Negli esempi che vengono mostrati inizialmente, si fa spesso uso della procedura '**`Writeln()`**', la quale si occupa semplicemente di emettere attraverso lo standard output tutti gli argomenti forniti. L'esempio seguente serve a emettere la frase «1000 volte ciao mondo!», utilizzando due parametri: la costante numerica 1000 e la stringa « volte ciao mondo!».

```
program CiaoMondo1000;

begin
    Writeln(1000, ' volte ciao mondo!');
end.
```

Si tenga presente, in ogni caso, che '**`Writeln`**' e '**`writeln`**' sono la stessa cosa.

Variabili e tipi

«

I tipi di dati elementari del linguaggio Pascal dipendono dal compilatore utilizzato e dall'architettura dell'elaboratore sottostante. I tipi standard del Pascal ANSI sono elencati nella tabella u119.4. Il tipo '**`char`**', non fa parte dello standard ANSI, ma è molto diffuso e così appare incluso in quella tabella.

Tabella u119.4. Elenco dei tipi di dati primitivi fondamentali in Pascal.

Tipo	Descrizione
`int`	Numeri interi positivi e negativi.
`byte`	Interi positivi di un solo byte (da 0 a 255).
`real`	Numeri a virgola mobile.
`boolean`	Valori logici booleani.
`char`	Carattere (generalmente di 8 bit).

Valori contenibili e costanti letterali

Ogni tipo di variabile può contenere un solo tipo di dati, esprimibile eventualmente attraverso una costante letterale scritta secondo una forma adatta.

I valori numerici vengono espressi da costanti letterali senza simboli di delimitazione.

- Gli interi (‘**`integer`**’) vanno espressi con numeri normali, senza punti di separazione di un’ipotetica parte decimale, prefissati eventualmente dal segno meno (‘**`-`**’) nel caso di valori negativi.
- I valori ‘**`byte`**’ vanno espressi come gli interi positivi, con la limitazione della dimensione massima.
- I numeri reali (‘**`real`**’) possono essere espressi come numeri aventi una parte decimale, segnalata dalla presenza di un punto decimale.

Se si vuole indicare un numero reale corrispondente a un numero intero, si deve aggiungere un decimale finto, per esempio, il numero 10 si può rappresentare come '**10.0**'.

Naturalmente è ammissibile anche la notazione esponenziale, come per esempio '**7e-2**' che corrisponde in pratica a 7*(10^{-2}), pari a 0,07 (scritto '**0.07**').

I valori logici vengono espressi dalle costanti letterali '**TRUE**' e '**FALSE**'.

I valori carattere e stringa, vengono delimitati da coppie di apici singoli, come ''**A**'', ''**B**'', ... ''**Ciao Mondo!**''.

Dichiarazione delle variabili

«

La dichiarazione delle variabili può essere fatta esclusivamente prima di un blocco '**begin**' '**end**' di un programma, di una funzione o di una procedura.

```
var nome : tipo;
```

Dalla sintassi si vede l'utilizzo della parola chiave '**var**', seguita dal nome della variabile da definire, quindi da due punti ('**:**'), infine dalla definizione del tipo di variabile.

In realtà, è possibile anche indicare un elenco di nomi, separati da virgole, quando questi devono essere tutti dello stesso tipo; inoltre, è possibile dichiarare più variabili differenti, utilizzando la parola chiave '**var**' una sola volta.

Segue la descrizione di alcuni esempi.

- ```
  var     conta   :       integer;
  ```
  ```

Dichiara la variabile '**conta**' di tipo intero.

- ```
  var     conta,canta     :       integer;
  ```

  Dichiara le variabili '**conta**' e '**canta**' di tipo intero.

- ```
  var     conta   :       integer;
          canta   :       integer;
  ```

  Esattamente uguale all'esempio precedente.

- ```
  var
          conta   :       integer;
          lettera :       char;
  ```

  Dichiara la variabile '**conta**' di tipo intero e la variabile '**lettera**' di tipo carattere.

Operatori ed espressioni

Gli operatori sono qualcosa che esegue un qualche tipo di funzione, su uno o due operandi, restituendo un valore. Il tipo di valore restituito varia a seconda dell'operatore e degli operandi utilizzati. Per esempio, la somma di due interi genera un intero, mentre una divisione di un valore intero per un altro numero intero, genera un numero reale.

Operatori aritmetici

Gli operatori che intervengono su valori numerici sono elencati nella tabella u119.9.

Tabella u119.9. Elenco degli operatori aritmetici e di quelli di assegnamento relativi a valori numerici.
  ```

Operatore e operandi	Descrizione
op1 `+` ***op2***	Somma i due operandi.
op1 `-` ***op2***	Sottrae dal primo il secondo operando.
op1 `*` ***op2***	Moltiplica i due operandi.
op1 `/` ***op2***	Divide il primo operando per il secondo, il risultato è in virgola mobile.
op1 `div` ***op2***	Divide il primo operando per il secondo generando un risultato intero.
op1 `mod` ***op2***	Modulo: il resto della divisione tra il primo e il secondo operando.
var `:=` ***valore***	Assegna alla variabile il valore alla destra.

Una caratteristica fondamentale del Pascal è la sua attenzione nella coerenza dei tipi di dati utilizzati nelle espressioni e negli assegnamenti. Tanto per comprendere il problema con un esempio, un compilatore non dovrebbe consentire l'assegnamento di un valore in virgola mobile in una variabile intera. Naturalmente, ogni compilatore può utilizzare una politica differente, consentendo una conversione di tipo automatica in situazioni particolari.

In ogni caso, è necessario conoscere l'uso di alcune funzioni essenziali, utili per prevenire conflitti nel tipo dei dati.

```
Round(numero_reale)
```

```
Trunc(numero_reale)
```

Le due funzioni, usate in questo modo, restituiscono un valore intero a partire da un valore a virgola mobile. Nel primo caso il numero viene arrotondato, mentre nel secondo viene semplicemente troncato al valore intero.

Operatori di confronto e operatori logici

Gli operatori di confronto determinano la relazione tra due operandi. Il risultato dell'espressione composta da due operandi messi a confronto è di tipo booleano, rappresentabile in Pascal con le costanti '**TRUE**' e '**FALSE**'. Gli operatori di confronto sono elencati nella tabella u119.10.

Tabella u119.10. Elenco degli operatori di confronto. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
op1 `=` ***op2***	*Vero* se gli operandi si equivalgono.
op1 `!=` ***op2***	*Vero* se gli operandi sono differenti.
op1 `<` ***op2***	*Vero* se il primo operando è minore del secondo.
op1 `>` ***op2***	*Vero* se il primo operando è maggiore del secondo.
op1 `<=` ***op2***	*Vero* se il primo operando è minore o uguale al secondo.
op1 `>=` ***op2***	*Vero* se il primo operando è maggiore o uguale al secondo.

Quando si vogliono combinare assieme diverse espressioni logiche, comprendendo in queste anche delle variabili che contengono un valore booleano, si utilizzano gli operatori logici. Gli operatori logici sono elencati nella tabella u119.11.

Tabella u119.11. Elenco degli operatori logici. Le metavariabili indicate rappresentano gli operandi e la loro posizione.

Operatore e operandi	Descrizione
`not` ***op***	Inverte il risultato logico dell'operando.
op1 `and` ***op2***	*Vero* se entrambi gli operandi restituiscono il valore *Vero*.
op1 `or` ***op2***	*Vero* se uno o entrambi gli operandi restituiscono il valore *Vero*.

Nel Pascal tradizionale, le espressioni logiche vengono valutate in ogni parte, prima di definire il risultato finale di un operatore AND o di un operatore OR. Dal momento che questo metodo di risoluzione è inutilmente dispersivo, spesso i compilatori Pascal consentono di ottenere il «cortocircuito», attraverso cui si valutano solo le parti dell'espressione che sono indispensabili per arrivare al risultato finale.

Strutture di controllo del flusso

«

Il linguaggio Pascal gestisce un buon numero di strutture di controllo di flusso, compreso il salto *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 un'istruzione singola, oppure un gruppo di istruzioni. Nel secondo caso, quasi sempre, è necessario delimitare questo gruppo attraverso l'uso di '**begin**' e '**end**'.

Dal momento che è comunque consentito di realizzare un gruppo di istruzioni che in realtà ne contiene una sola, probabilmente è meglio utilizzare sempre i delimitatori '**begin**' '**end**', a vantaggio dello stile e della leggibilità del codice.

Struttura condizionale: «if»

```
if condizione then istruzione
```

```
if condizione then istruzione else istruzione
```

Se la condizione si verifica, viene eseguita l'istruzione (o il gruppo di istruzioni) seguente; quindi il controllo passa alle istruzioni successive alla struttura. Se viene utilizzato '**else**', nel caso non si verifichi la condizione, viene eseguita l'istruzione che ne segue. Vengono mostrati alcuni esempi.

```
…
var     importo : integer;
…
if importo > 10000000 then Writeln( 'offerta vantaggiosa' );
```

```
…
var     importo          : integer;
        memorizza        : integer;
…
if importo > 10000000 then
begin
    memorizza := importo;
    Writeln( 'offerta vantaggiosa' );
end
else
    Writeln( 'meglio lasciar perdere' );
```

```
…
var     importo          : integer;
        memorizza        : integer;
…
if importo > 10000000 then
begin
    memorizza := importo;
    Writeln( 'offerta vantaggiosa' );
end
else if importo > 5000000 then
begin
    memorizza := importo;
    Writeln( 'offerta accettabile' );
end
else
    Writeln( 'meglio lasciar perdere' );
```

Il blocco *if-then-else* rappresenta un'unica istruzione in Pascal. In questo senso, dovrebbe apparire un punto e virgola alla fine del blocco, a terminare l'istruzione. Se si utilizzano raggruppamenti di istruzioni attraverso i delimitatori '**begin**' '**end**', le istruzioni contenute terminano con il punto e virgola, mentre il blocco, dopo la parola chiave '**end**', no, a meno che si tratti della fine dell'istruzione '**if**'.

Per osservare meglio questo particolare, si potrebbero riscrivere gli

stessi esempi nel modo seguente, in cui il punto e virgola finale serve a concludere visivamente la dentellatura delle istruzioni '**`if`**'.

```
…
var     importo : integer;
…
if importo > 10000000 then
    Writeln( 'offerta vantaggiosa' )
;
```

```
…
var     importo         : integer;
        memorizza       : integer;
…
if importo > 10000000 then
    begin
        memorizza := importo;
        Writeln( 'offerta vantaggiosa' );
    end
else
    Writeln( 'meglio lasciar perdere' )
;
```

```
…
var     importo         : integer;
        memorizza       : integer;
…
if importo > 10000000 then
    begin
        memorizza := importo;
        Writeln( 'offerta vantaggiosa' );
    end
else
    if importo > 5000000 then
        begin
            memorizza := importo;
            Writeln( 'offerta accettabile' );
        end
    else
        Writeln( 'meglio lasciar perdere' )
;
```

Struttura di selezione: «case»

«

La struttura di selezione si ottiene con l'istruzione '**case**'. Si tratta di una struttura un po' troppo complessa per essere rappresentata facilmente attraverso uno schema sintattico. In generale, l'istruzione '**case**' 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.

```
…
var     mese    : integer;
…
case mese of
    1  : Writeln( 'gennaio' );
    2  : Writeln( 'febbraio' );
    3  : Writeln( 'marzo' );
    4  : Writeln( 'aprile' );
    5  : Writeln( 'maggio' );
    6  : Writeln( 'giugno' );
    7  : Writeln( 'luglio' );
    8  : Writeln( 'agosto' );
    9  : Writeln( 'settembre' );
    10 : Writeln( 'ottobre' );
    11 : Writeln( 'novembre' );
    12 : Writeln( 'dicembre' );
end;
```

È importante osservare l'uso del punto e virgola, che conclude ogni istruzione richiamata dai vari casi. La parola chiave '**end**' finale, conclude la struttura.

Un gruppo di casi può essere raggruppato assieme, quando si vuole che ognuno di questi esegua lo stesso insieme di istruzioni:

```
…
var     anno    : integer;
        mese    : integer;
        giorni  : integer;
…
case mese of
    1,3,5,7,8,10,12 :
        giorni := 31;
    4,6,9,11 :
        giorni := 30;
    2   :
        if ((anno mod 4 = 0) and not (anno mod 100 = 0)) or
                (iAnno mod 400 = 0) then
            giorni := 29
        else
            giorni := 28
        ;
end;
```

È anche possibile definire un caso predefinito che si verifichi quando nessuno degli altri si avvera:

```
…
var     mese    : integer;
…
case mese of
    1  : Writeln( 'gennaio' );
    2  : Writeln( 'febbraio' );
…
    11 : Writeln( 'novembre' );
    12 : Writeln( 'dicembre' );
else
    Writeln( 'mese non corretto' );
end;
```

Un intervallo di casi può essere indicato facilmente come nell'esempio seguente:

```
…
var     mese    : integer;
…
case mese of
    6..9 : Writeln( 'mesi caldi' );
    …
end;
```

Iterazione con condizione di uscita iniziale: «while»

«

```
while condizione do istruzione
```

La struttura '**while**' esegue un'istruzione finché la condizione restituisce il valore *Vero*. La condizione viene valutata prima di eseguire l'istruzione e poi ogni volta che termina un ciclo, prima dell'esecuzione del successivo.

Come sempre, al posto della singola istruzione se ne può inserire un raggruppamento, delimitato dalle parole chiave '**begin**' e '**end**'. L'esempio seguente fa apparire per 10 volte la lettera «x»:

```
program DieciX;

var contatore   : integer;

begin
    contatore := 0;
    while contatore < 10 do
    begin
        contatore := contatore + 1;
        Writeln( 'x' );
    end;
end.
```

La struttura '**while**' è un'istruzione singola in Pascal. Per sottolinearlo, si potrebbe cambiare la dentellatura dell'esempio appena

mostrato per fare in modo che il punto e virgola finale, che chiude l'istruzione, inizi sulla stessa colonna della parola chiave '**while**'.

```
…
    contatore := 0;
    while contatore < 10 do
        begin
            contatore := contatore + 1;
            Writeln( 'x' );
        end
    ;
…
```

Iterazione con condizione di uscita finale: «repeat-until»

«

```
repeat istruzione ; … until condizione ;
```

La struttura '**repeat**' '**until**' permette di eseguire un gruppo di istruzioni una volta e poi di ripeterne l'esecuzione fino a quando la condizione posta alla fine continua a non verificarsi.

Ci sono quindi due diversità fondamentali, rispetto alla struttura '**while**': il gruppo di istruzioni viene eseguito sicuramente almeno una volta; il verificarsi della condizione implica l'interruzione del ciclo.

Per quanto riguarda la sintassi usata dal Pascal, c'è da osservare che dopo la parola chiave '**repeat**' possono essere collocate una serie di istruzioni, senza bisogno di un raggruppamento '**begin**' '**end**'. In questo senso, ogni istruzione termina con il suo punto e virgola.

L'esempio seguente è solo un pretesto per mostrare il funzionamento di questa struttura: visualizza 10 volte la lettera «x».

```
program DieciX;

var contatore   : integer;

begin
    contatore := 0;
    repeat
        contatore := contatore + 1;
        Writeln( 'x' );
    until contatore = 10;
end.
```

Iterazione enumerativa: «for»

«

```
for variabile; := inizio to fine do istruzione
```

L'istruzione '**for**' permette di definire un ciclo enumerativo, in cui una variabile intera viene inizializzata, quindi viene eseguita ripetitivamente l'istruzione controllata, incrementando alla fine di ogni esecuzione tale variabile e interrompendo il ciclo quando questa raggiunge il valore finale (quando la variabile ha raggiunto il valore finale, si esegue l'istruzione per l'ultima volta). L'incremento è di un'unità quando il valore finale è maggiore di quello iniziale, oppure di un'unità negativa quando il valore finale è minore di quello iniziale.

L'esempio già visto, in cui veniva visualizzata per 10 volte una «x», potrebbe tradursi nel modo seguente, attraverso l'uso di un ciclo '**for**'.

```
program DieciX;

var contatore   : integer;

begin
    for contatore := 1 to 10 do
        Writeln( 'x' )
    ;
end.
```

Come sempre, al posto di controllare una singola istruzione, se ne può gestire un gruppo, attraverso l'uso dei delimitatori '**begin**' e '**end**'. L'esempio già visto, potrebbe eventualmente tradursi nel modo seguente:

```
…
    for contatore := 1 to 10 do
        begin
            Writeln( 'x' );
        end
    ;
…
```

Procedure e funzioni

Il linguaggio Pascal distingue due tipi di subroutine: procedure e funzioni. In pratica, le procedure sono funzioni che non restituiscono alcun valore.

La dichiarazione e descrizione delle procedure e delle funzioni deve essere fatta all'interno della parte iniziale del programma, dedicata alle dichiarazioni. Procedure e funzioni possono chiamarsi a vicenda e, in ogni caso, perché la chiamata possa essere valida, occorre che la procedura o la funzione sia stata dichiarata precedentemente.

Ci sono situazioni in cui non è possibile descrivere una funzione o una procedura prima di quella chiamante. In tali casi, è possibile

dichiarare una funzione senza descriverla immediatamente.

Struttura

«

Per il linguaggio Pascal, le procedure e le funzioni sono dei sottoprogrammi veri e propri, tanto che anche in questo caso si distinguono tre parti: intestazione, dichiarazioni e istruzioni. In particolare, l'intestazione può includere anche la dichiarazione, a meno che questa non sia separata per renderla visibile ad altre procedure e funzioni precedenti.

```
procedure nome [ (parametro_formale [...] ) ] ;
```

```
function nome [ (parametro_formale [...] ) ]  : tipo ;
```

La sintassi che appare sopra rappresenta la dichiarazione di una procedura e di una funzione. Come si può osservare, a parte la parola chiave iniziale, la funzione ha alla fine l'indicazione del tipo di dati che restituisce.

Se la procedura o la funzione non richiede l'indicazione di parametri, allora non è necessario specificare alcun ***parametro formale***, quindi non sono necessarie nemmeno le parentesi tonde.

Dopo la dichiarazione della funzione o della procedura, vanno indicate le dichiarazioni, per esempio le variabili utilizzate, nello stesso modo già visto per il programma.

Infine vanno poste le istruzioni, all'interno di un raggruppamento '**`begin`**' '**`end`**'. A differenza del raggruppamento analogo che ri-

guarda il blocco principale del programma, la parola chiave '**end**' è conclusa con un punto e virgola invece che con il punto.

La funzione restituisce un valore, attraverso l'assegnamento a una variabile ipotetica che ha lo stesso nome della funzione.

Segue la descrizione di alcuni esempi.

- ```
  procedure CiaoCiao;
  begin
      Writeln('Ciao a tutti');
      Writeln('ciao ciao ciao');
  end;
  ```
  Si tratta di una procedura elementare che non utilizza alcun parametro e si limita a emettere un messaggio di saluto.

- ```
  function CiaoCiao : boolean;
  begin
      Writeln('Ciao a tutti');
      Writeln('ciao ciao ciao');
      CiaoCiao := TRUE;
  end;
  ```
  Si tratta di una funzione elementare che non utilizza alcun parametro e si limita a emettere un messaggio di saluto, restituendo sempre il valore booleano *Vero*.

Campo di azione

Sia le variabili che le procedure e le funzioni, hanno un campo di azione. Le variabili dichiarate nella parte introduttiva di un programma, prima della dichiarazione di procedure e funzioni, sono accessibili al corpo del programma e a tutte le procedure e funzioni. Le variabili dichiarate nella parte introduttiva di una procedura o di una funzione, hanno effetto locale, non essendo visibili all'esterno; se queste hanno nomi già utilizzati per le variabili globali, di fatto ne impediscono l'accesso.

Le procedure e le funzioni, in qualità di sottoprogrammi, possono contenere anche la dichiarazione di sottoprocedure e sottofunzioni. In tal caso, tali subroutine sono accessibili solo dal codice contenuto nella procedura o funzione in cui sono dichiarate. Nello stesso modo, le variabili locali delle procedure o delle funzioni sono accessibili anche alle rispettive sottoprocedure e sottofunzioni.

Forward

«

Si è accennato al fatto che, perché una chiamata possa essere valida, occorre che la procedura o la funzione in questione sia stata dichiarata prima, cioè in una posizione precedente all'interno del sorgente.

In presenza di chiamate ricorsive tra più procedure o funzioni, diviene impossibile che ogni chiamata si riferisca sempre a qualcosa di definito e descritto in precedenza.

Per risolvere il problema, si può dichiarare una procedura o una funzione prima della sua descrizione effettiva, attraverso l'uso della parola chiave '**`forward`**', come nell'esempio seguente:

```
…
procedure MiaProcedura(…);
forward;
…
…
procedure MiaProcedura;
begin
    …
end;
…
```

La dichiarazione della procedura o della funzione deve contenere la dichiarazione di tutti i parametri formali, mentre la descrizione è assente.

La descrizione dei parametri formali, all'interno della dichiarazione di una procedura o di una funzione, richiede la definizione del nome delle variabili e del tipo relativo. Il campo di azione di queste variabili è locale.

```
…
procedure MiaProcedura( primo,secondo : integer;
                        terzo         : char);
begin
    …
end;
…
```

L'esempio mostra la dichiarazione di una procedura che utilizza tre parametri formali, denominati casualmente proprio: '**primo**', '**secondo**' e '**terzo**'. I primi due sono di tipo '**integer**', mentre l'ultimo è di tipo '**char**'.

Come si può osservare, la dichiarazione dei parametri formali è molto simile alla dichiarazione delle variabili, con la differenza che ciò avviene all'interno di parentesi tonde, oltre al fatto che (per il momento) manca la parola chiave '**var**'.

Una procedura o una funzione in cui i parametri formali siano stati dichiarati in questo modo, riceve una copia dei dati nel momento della chiamata, senza poter riflettere all'indietro le modifiche che a questi dovesse applicare. Si ha in pratica una chiamata per valore.

È possibile dichiarare una procedura o una funzione in cui la chiamata sia per riferimento, in modo da riflettere all'indietro le modifiche, utilizzando la parola chiave '**var**'.

```
…
procedure MiaProcedura( primo       : integer;
                        var secondo : integer;
                        terzo       : char);
begin
    …
end;
…
```

L'esempio mostra una variante in cui si dichiara che il secondo parametro formale, '**secondo**', riflette all'indietro le modifiche che dovessero essergli apportate all'interno della procedura.

Chiamata e parametri attuali

«

La chiamata di una procedura o di una funzione, avviene semplicemente nominandola e facendola seguire dall'indicazione dei ***parametri attuali***, cioè dei valori che si vuole siano passati per l'elaborazione.

La differenza fondamentale tra procedure e funzioni sta nel fatto che le chiamate alle prime vengono utilizzate come istruzioni pure e semplici, mentre le seconde, vanno inserite all'interno di espressioni.

Merita un minimo di attenzione anche il tipo di chiamata: per valore o per riferimento. Nel primo caso, non si pongono problemi di alcun tipo, dal momento che la funzione o la procedura chiamata non può alterarli; se invece si tratta di una chiamata per riferimento, occorre fare attenzione che il parametro attuale, usato nella chiamata, non sia una costante, perché questo genererebbe un errore irreversibile.

```
…
var     MioNumero : integer;
…
procedure MiaProcedura( primo       : integer;
                        var secondo : integer;
                        terzo       : char);
begin
    …
    secondo := 777;
    …
end;
…
{ inizio del programma }
begin
    MiaProcedura( 123, MioNumero, 'C' );
    Writeln( MioNumero );
end.
```

L'esempio mostra una chiamata a una procedura in cui uno dei parametri deve essere chiamato per riferimento. In tal caso, il parametro attuale corrispondente, utilizzato nella chiamata, è necessariamente una variabile.

I/O elementare

Per le operazioni di I/O elementare, cioè per l'utilizzo di standard output e standard error, si hanno a disposizione due coppie di procedure: '**`Write()`**' e '**`Writeln()`**'; '**`Read()`**' e '**`Readln()`**'. La prima coppia per emettere qualcosa attraverso lo standard output, la seconda per leggere qualcosa dallo standard input.

Anche se non è ancora stato affrontato l'argomento stringhe, è opportuno anticipare che per inserire un apice singolo all'interno di una costante stringa, basta indicarne due consecutivi. Per esempio, la stringa seguente,

```
'questa è la ''vera'' verità'
```

Si traduce in:

```
questa è la 'vera' verità
```

Procedure «Write()» e «Writeln()»

«

```
Write(elemento_da_visualizzare [ :dimensione [ :decimali ] ] [,…] )
```

```
Writeln(elemento_da_visualizzare [ :dimensione [ :decimali ] ] [,…] )
```

Le procedure '**`Write()`**' e '**`Writeln()`**' permettono di emettere attraverso lo standard output il contenuto di tutti i parametri che gli vengono forniti. A seconda dei tipi di dati utilizzati, vengono effettuate tutte le conversioni necessarie a ottenere un risultato stringa.

Se un parametro attuale, fornito nella chiamata, viene indicato seguito da due punti ('`:`') e quindi da un numero, si stabilisce lo spazio (espresso in colonne) che questo deve utilizzare nell'output. Se si specifica tale dimensione, l'informazione viene rappresentata allineandola a destra. Questa possibilità di definire la dimensione viene utilizzata prevalentemente per i dati numerici e in questo senso sta la logica dell'allineamento a destra.

Se si vuole rappresentare un valore numerico con decimali, è abbastanza importante fissare la dimensione della visualizzazione, aggiungendo anche l'indicazione delle colonne da riservare alla parte decimale. Diversamente, la rappresentazione risulterebbe in notazione esponenziale.

L'unica differenza tra le due procedure, sta nel fatto che '**`Writeln()`**' aggiunge automaticamente, alla fine della stringa visualizzata, il codice di interruzione di riga, in modo da riportare il cursore all'inizio della riga successiva.

Segue la descrizione di alcuni esempi.

- ```
  var totale : integer;
  …
  totale := 1950000;
  …
  Write('Totale:', totale:11);
  ```

  Emette la stringa seguente, senza portare a capo il cursore alla fine:

  ```
  Totale:    1950000
  ```

- ```
  var totale : real;
  …
  real := 1234.5678;
  …
  Writeln('Totale:', totale:11:5);
  ```

  Emette la stringa seguente, portando a capo il cursore alla fine.

  ```
  Totale: 1234.56780
  ```

Procedure «Read()» e «Readln()»

«

```
Read(variabile [,…])
```

```
Readln(variabile [,…])
```

Le procedure '**`Read()`**' e '**`Readln()`**' permettono di leggere dallo standard input dei valori per le variabili che vengono indicate come

parametri della chiamata. I dati inseriti, vengono distinti in base all'inserimento di spaziature, così come avviene di solito con gli argomenti di un comando del sistema operativo.

È importante che i dati inseriti siano compatibili con il tipo delle variabili utilizzate, altrimenti si rischia di ottenere un errore irreversibile durante il funzionamento del programma.

La differenza tra le due procedure sta nel fatto che '**`Readln()`**' dovrebbe restituire l'eco del codice di interruzione di riga, quando si preme [*Invio*] per concludere l'inserimento dei dati, mentre '**`Read()`**' no. In pratica, può darsi che il compilatore non riesca a distinguere tra le due procedure, comportandosi sempre nello stesso modo.

Segue la descrizione di alcuni esempi.

- ```
  var totale : integer;
  …
  Write('Inserisci il totale: ');
  Read(totale);
  …
  ```

  Emette l'invito a inserire un valore e quindi lo attende dallo standard input.

- ```
  var capitale : integer;
  var tasso    : real;
  …
  Write('Inserisci di seguito il capitale e il tasso: ');
  Read(capitale,tasso);
  …
  ```

  Emette l'invito a inserire due valori consecutivi: un intero e un valore decimale.

Struttura del sorgente: le dichiarazioni

È già stato accennato alla struttura di un sorgente Pascal: del programma, delle procedure e delle funzioni. Si tratta di tre parti fondamentali:

1. intestazione del programma, dichiarazione della procedura o della funzione;
2. dichiarazioni;
3. istruzioni.

Il punto più delicato è la definizione della parte delle dichiarazioni, dato che nel Pascal originale esiste un ordine preciso nel tipo di istruzioni che possono esservi inserite. Si tratta di dichiarazioni:

1. '**`label`**'
2. '**`const`**'
3. '**`type`**'
4. '**`var`**'
5. '**`procedure`**'
6. '**`function`**'

La maggior parte di queste dichiarazioni non è ancora stata descritta. In particolare, '**`label`**', dal momento che serve a realizzare dei salti incondizionati senza ritorno (*go-to*), non viene descritta in questi capitoli sul Pascal.

«

- Gordon Dodrill, *Pascal Language Tutorial*
  http://www.geocities.com/hotdogcom/ptutor/paslist.html
  http://packetstormsecurity.nl/programming-tutorials/Pascal/pascal-tutorial/paslist.htm

Pascal: tipi di dati derivati

Nel capitolo introduttivo è stato visto l'uso di variabili identificabili semplicemente con il loro nome. La programmazione elementare richiede anche l'utilizzo di strutture di dati più complesse; le stesse stringhe sono degli array di caratteri e come tali vanno trattate.

Array

Gli array in Pascal sono una sequenza ordinata, in una quantità prestabilita, di elementi dello stesso tipo. Gli elementi possono essere composti da qualunque tipo di dati, nativo o derivato.

Una caratteristica importante del linguaggio Pascal sta nel fatto che nel momento della dichiarazione di un array, viene definito anche il valore iniziale dell'indice da utilizzare per la scansione dei vari elementi.

Dichiarazione e accesso

```
var nome : array[inizio..fine] of tipo
```

La sintassi indicata, dove le parentesi quadre fanno parte dell'istruzione, mostra in breve in che modo si possa dichiarare un array, a una sola dimensione, di elementi di un certo tipo di dati.

È importante osservare che vengono stabiliti in modo esplicito sia l'indice iniziale del primo elemento, sia quello finale dell'ultimo, stabilendo implicitamente la quantità di questi elementi.

L'esempio seguente mostra la dichiarazione di tre array simili, composti tutti da sette interi, dove, rispettivamente, il primo elemento si raggiunge con l'indice iniziale 1, 0 e 2.

```
var elenco  : array[1..7] of integer;
    elenco2 : array[0..6] of integer;
    elenco3 : array[2..8] of integer;
```

Per accedere agli elementi di un array si usa la sintassi seguente e anche qui le parentesi quadre fanno parte dell'istruzione.

```
nome [indice]
```

Quello che conta è che l'indice indicato sia valido, in funzione della dichiarazione fatta in origine. L'esempio seguente assegna al primo elemento il valore 10.

```
elenco[1] := 10;
```

Gli array multidimensionali non sono altro che array di array. Il modo più semplice per dichiarare un array multidimensionale è quello di indicare due o più intervalli di valori per gli indici, secondo la sintassi seguente:

```
var nome : array[inizio . .fine , inizio . .fine ...] of tipo
```

Per esempio, l'istruzione seguente dichiara un array a due dimensioni di tre elementi per otto, di tipo intero. Si osservi in particolare il secondo intervallo di indici, dove il primo elemento viene raggiunto con l'indice zero.

```
var elenco : array[1..3,0..7] of integer;
```

In modo analogo, si raggiunge un elemento di un array multidimensionale utilizzando due o più indici, secondo la sintassi seguente:

```
nome [indice , indice ...]
```

L'esempio seguente assegna un valore all'elemento «1,0».

```
elenco[1,0] := 10;
```

Scansione di un array

«

La scansione di un array avviene generalmente con un ciclo enumerativo, '**`for`**', come nell'esempio seguente:

```
…
var indice : integer;
var elenco : array[1..7] of integer;
…
begin
    …
    for indice := 1 to 7 do begin
        …
        elenco[indice] := …
        …
    end;
    …
end.
```

La scansione di array multidimensionali avviene generalmente attraverso una serie di cicli enumerativi, uno per ogni dimensione, annidati opportunamente. L'esempio seguente mostra la scansione di un array a tre dimensioni.

```
…
var i,j,k : integer;
var elenco : array[1..7,0..8,2..10] of integer;
…
begin
    …
    for i := 1 to 7 do begin
        …
        for j := 0 to 8 do begin
            …
            for k := 2 to 10 do begin
                …
                elenco[i,j,k] := …
                …
            end;
            …
        end;
        …
    end;
    …
end.
```

Stringhe

Nel linguaggio Pascal, così come in molti altri, le stringhe sono semplicemente degli array di caratteri, con qualche piccola differenza per facilitarne l'utilizzo.

La dichiarazione di una variabile stringa è quindi la dichiarazione di un array composto da una quantità predefinita di caratteri. Nell'esempio seguente, viene creato una variabile stringa di 20 caratteri.

```
var cognome : array[1..20] of char;
```

La variabile dichiarata in questo modo può essere usata come un array, cioè accedendo alle informazioni carattere per carattere, oppure nel suo insieme. Nell'esempio seguente si assegna un nome alla

variabile stringa mostrata sopra.

```
cognome := 'Rossi';
```

Se si utilizza un assegnamento di questo tipo, vengono ricoperti anche gli elementi successivi alla lunghezza della stringa letterale assegnata. Quindi, seguendo l'esempio, l'array riceve il nome «Rossi» nei suoi primi cinque elementi, mentre negli altri viene comunque inserito uno spazio.

Tipi

«

Il linguaggio Pascal permette di definire dei tipi di dati derivati, a partire da quelli elementari, o a partire da altri tipi composti dichiarati precedentemente.

```
type tipo_nuovo = definizione_del_tipo
```

La definizione di un nuovo tipo va posta nella zona dichiarativa del programma, della procedura o della funzione. L'esempio seguente serve a dichiarare il tipo «Numero» come equivalente al tipo intero standard.

```
type Numero = integer;
```

Naturalmente, la definizione di un nuovo tipo è sensata quando serve a individuare qualcosa di più complesso dei dati elementari, come nel caso di un array. L'esempio seguente dichiara il tipo «Stringa» come un array di 80 caratteri, quindi dichiara il tipo «Nominativo» come array composto da due elementi «Stringa» (probabilmente uno per il nome e l'altro per il cognome).

```
type Stringa = array[1..80] of char;
type Nominativo = array[1..2] of Stringa;
```

A questo punto, per seguire l'esempio, se si generasse una variabile di tipo «Nominativo», si otterrebbe un array di due elementi, che in realtà sono array di 80 caratteri.

```
…
var Nome : Nominativo;
…
begin
    …
    Nome[1] := 'Pinco';
    Nome[2] := 'Pallino';
    …
end.
```

L'esempio mostra in che modo si potrebbe usare una variabile del genere. Tuttavia, si potrebbe accedere anche al singolo elemento carattere, utilizzando due indici.

```
    …
    Nome[1,1] := 'P';
    Nome[1,2] := 'i';
    Nome[1,3] := 'n';
    Nome[1,4] := 'c';
    Nome[1,5] := 'o';
    …
end.
```

Convenzionalmente, quando si dichiara un nuovo tipo di dati, si usa l'iniziale maiuscola, per distinguerlo facilmente dagli altri tipi nativi.

Costanti

Il linguaggio Pascal offre qualcosa di simile alle costanti macro di altri linguaggi come il C. Non si tratta di un linguaggio di precompilazione, ma proprio del Pascal, anche se si tratta comunque di costanti letterali, senza la definizione di un tipo a priori.

```
const nome_della_costante = valore_letterale
```

La dichiarazione di queste costanti va fatta, come prevedibile, nella zona dichiarativa del programma, della procedura o della funzione. L'esempio seguente dichiara la costante «DIMENSIONE», che poi viene usata per definire la dimensione di una serie di array.

```
…
const DIMENSIONE = 11;
…
var elenco  : array[1..DIMENSIONE] of integer;
    elenco2 : array[1..DIMENSIONE] of integer;
    elenco3 : array[1..DIMENSIONE] of integer;
```

Il vantaggio di utilizzare le costanti sta nel facilitare la lettura del sorgente, nel riconoscere il significato di determinate costanti e nel facilitare la modifica di tali valori, senza dover rileggere tutto il sorgente alla loro ricerca.

Tipo enumerativo, sottointervallo e insieme

«

Il linguaggio Pascal offre dei tipi di dati particolari, che non sono ancora stati descritti, il cui scopo è solo quello di facilitare il compito del programmatore.

Tipo enumerativo

«

Il tipo enumerativo, o scalare, secondo la terminologia del Pascal, è una forma di rappresentazione di un intero attraverso costanti mnemoniche. In pratica, si definisce una variabile che può assumere un elenco di valori simbolici possibili, valori che in realtà sono solo delle costanti simboliche, senza un legame con il termine letterale usato per distinguerle, salva la convenienza del programmatore.

(*costante* , *costante* [, …])

La sintassi indicata mostra il modo in cui si definisce un tipo del genere: all'interno di parentesi tonde si elencano i nomi delle costanti che possono essere assegnate a una variabile di questo tipo.

L'esempio seguente mostra la dichiarazione di una variabile scalare che può assumere i valori «VERDE», «BLU» e «ROSSO».

```
var colore : (VERDE, BLU, ROSSO);
```

L'esempio stesso dovrebbe chiarire l'utilità di questo tipo di dati: si lascia al compilatore il compito di stabilire i valori più appropriati per i simboli che possono essere associati a una variabile. Tuttavia, è importante chiarire che non è possibile visualizzare il contenuto di una variabile del genere, in quanto questo non è prevedibile.

```
if colore = VERDE then
    begin
        …
        Writeln( "Il colore è verde" );
    end;
else
    …
;
```

Naturalmente, questo tipo di dati si presta particolarmente per la definizione di tipi derivati, come nell'esempio seguente, dove prima si dichiara un tipo e più avanti si utilizza nella dichiarazione di una nuova variabile.

```
type Sapore = (INSIPIDO, DOLCE, SALATO, ACIDO, PICCANTE, AMARO);
…
var pietanza : Sapore;
…
```

Sottointervallo

«

Il sottointervallo è la definizione di un tipo derivato che può utilizzare solo un intervallo stabilito di valori. Questo intervallo si definisce solo con l'indicazione di due costanti dello stesso tipo, separate da due punti in sequenza.

Per esempio, per indicare la serie di numeri interi che va da uno a sette, si può utilizzare la notazione '**1..7**', mentre per indicare la serie delle lettere alfabetiche minuscole, si può utilizzare la notazione '**'a'..'z'**'.

Naturalmente, si possono indicare anche degli intervalli di un tipo enumerativo dichiarato in precedenza. Seguono alcuni esempi.

```
type Settimana = (LUNEDÌ, MARTEDÌ, MERCOLEDÌ,
                  GIOVEDÌ, VENERDÌ, SABATO, DOMENICA);

type Feriale   = LUNEDÌ..VENERDÌ;
…
var lavoro    : Feriale;
    minuscola : 'a'..'z';
…
```

Le variabili dichiarate in questo modo, ottengono dal compilatore il tipo più adatto a contenere l'informazione indicata, senza la necessità di doverlo indicare in modo esplicito.

Insieme

«

Una variabile può contenere un'informazione riferita a un insieme di elementi enumerativi. In pratica, si tratta di un tipo simile a quello enumerativo, dove ogni elemento può essere presente o meno. Si dichiara questo tipo di dati con le parole chiave '**set of**'. Si osservi l'esempio seguente:

```
type Settimana = (LUNEDÌ, MARTEDÌ, MERCOLEDÌ,
                  GIOVEDÌ, VENERDÌ, SABATO, DOMENICA);
…
type Lavoro    = set of Settimana;
…
var tutti     : Lavoro;
    presenze  : Lavoro;
    assenze   : Lavoro;
    altri     : Lavoro;
…
```

Le variabili '**tutti**', '**presenze**' e '**assenze**', definite del tipo '**Lavoro**', il quale a sua volta è definito come insieme di tutti i simboli del tipo '**Settimana**', possono contenere un sottoinsieme di tali simboli.

```
…
begin
    …
    presenze := (LUNEDÌ, MERCOLEDÌ, VENERDÌ,
                 DOMENICA);
    …
    tutti := (LUNEDÌ..DOMENICA);
    …
    assenze := tutti - presenze;
    …
    altri := assenze;
    …
    tutti := assenze + presenze;
    …
end.
```

L'esempio mostra alcuni modi in cui possono essere utilizzate le variabili contenenti insiemi e quali espressioni si possono realizzare. In pratica:

- due variabili dello stesso tipo di insieme possono essere assegnate l'una nell'altra;
- due variabili dello stesso tipo di insieme possono essere sommate,

generando un insieme risultato dell'unione dei due;

- tra due variabili dello stesso tipo di insieme può essere indicata una sottrazione, con la quale si genera un insieme risultato dall'eliminazione degli elementi presenti nella seconda variabile.

A parte gli assegnamenti che possono essere fatti alle variabili contenenti un insieme, è poi necessario poter verificare il contenuto di tali variabili, con istruzioni apposite. Per questo si usa la parola chiave '**in**'. L'esempio seguente dovrebbe essere autoesplicativo.

```
    if LUNEDÌ in presenze then begin
        …
        …
    end;
    if MARTEDÌ in presenze then begin
        …
        …
    end;
```

Un insieme può essere definito anche come gruppo di valori di un intervallo, come nell'esempio seguente in cui si definisce un tipo nuovo che rappresenta l'insieme delle lettere minuscole.

```
type Lettere    = set of 'a'..'z';
```

Nello stesso modo, si può utilizzare la parola chiave '**in**' per verificare che un valore appartenga a un insieme definito in forma di intervallo.

```
if iniziale in 'a'..'z' then begin
    …
end;
```

Record

Il record è un tipo di dati composto dall'insieme di altri tipi, ognuno con una sua denominazione. L'esempio seguente mostra in che modo possano essere creati tipi nuovi definiti come record.

```
type Datario =
    record
        anno    : integer;
        mese    : integer;
        giorno  : integer;
    end;

type Anagrafico =
    record
        cognome : array[1..40] of char;
        nome    : array[1..40] of char;
        luogo   : array[1..40] of char;
        data    : Datario;
    end;
```

L'esempio vuole mostrare la creazione di un record anagrafico con tutti i dati (riferiti alla nascita) che permettono di identificare una persona. Si può osservare che la data (di nascita) è stata definita come tipo '**`Datario`**', che a sua volta è un altro record.

Quando si dichiara una variabile come tipo record, si pone il problema di accedere ai vari elementi di questo. Per farlo si usa l'operatore punto ('`.`'). Si osservi l'esempio seguente, in cui si dichiara un array di dati anagrafici e quindi si assegnano i valori per il primo elemento di questo array.

```
…
var anagrafe : array[1..10] of Anagrafico;
…
begin
    …
    anagrafe[1].cognome         := 'Pallino';
    anagrafe[1].nome            := 'Pinco';
    anagrafe[1].luogo           := 'Sferopoli';
    anagrafe[1].data.anno       := 1990;
    anagrafe[1].data.mese       := 1;
    anagrafe[1].data.giorno     := 31;
    …
end;
```

Come si può osservare, per inserire le informazioni sulla data di nascita, è stato necessario usare due volte il punto per accedere agli elementi del sottorecord '**`data`**'.

Una variabile definita come record può ricevere l'assegnamento in blocco di un'altra variabile record, purché dello stesso tipo.

With

«

Quando si utilizzano frequentemente i record, potrebbe essere conveniente specificare che in una porzione di codice sorgente si vuole fare riferimento a elementi di una variabile determinata. Si osservi l'esempio seguente, che è una variante di quanto già visto in precedenza.

```
…
var anagrafe : array[1..10] of Anagrafico;
…
begin
    …
    with anagrafe[1] do begin
        cognome          := 'Pallino';
        nome             := 'Pinco';
        luogo            := 'Sferopoli';
        data.anno        := 1990;
        data.mese        := 1;
        data.giorno      := 31;
    end;
    …
end;
```

Il significato dovrebbe essere evidente: nell'intervallo delimitato dalle parole chiave '**`begin`**' '**`end`**', tutti i nomi si riferiscono a elementi di '**`anagrafe[1]`**'.

Riferimenti

«

- Gordon Dodrill, *Pascal Language Tutorial*
  http://www.geocities.com/hotdogcom/ptutor/paslist.html
  http://packetstormsecurity.nl/programming-tutorials/Pascal/pascal-tutorial/paslist.htm

Pascal: esempi di programmazione

Questo capitolo raccoglie solo alcuni esempi di programmazione, in parte già descritti in altri capitoli. Lo scopo di questi esempi è solo didattico, utilizzando forme non ottimizzate per la velocità di esecuzione.

Problemi elementari di programmazione

«

In questa sezione vengono mostrati alcuni algoritmi elementari portati in Pascal.

Somma tra due numeri positivi

Il problema della somma tra due numeri positivi, attraverso l'incremento unitario, è descritto nella sezione 62.3.1.

```
(* ================================================================ *)
(* Somma.pas                                                        *)
(* Somma esclusivamente valori positivi.                            *)
(* ================================================================ *)
program Sommare;

var     x       : integer;
        y       : integer;
        z       : integer;

(* ================================================================ *)
(* somma( <x>, <y> )                                                *)
(* ---------------------------------------------------------------- *)
function somma( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := x;

    for i := 1 to y do begin
        z := z+1;
    end;

    somma := z;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
```

```
    Readln( y );

    z := somma( x, y );

    Write( x, ' + ', y, ' = ', z );

end.

(* ================================================================= *)
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**':

```
function somma( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := x;
    i := 1;

    while i <= y do begin
        z := z+1;
        i := i+1;
    end;

    somma := z;

end;
```

Moltiplicazione di due numeri positivi attraverso la somma

«

Il problema della moltiplicazione tra due numeri positivi, attraverso la somma, è descritto nella sezione 62.3.2.

```
(* ================================================================= *)
(* Moltiplica.pas                                                    *)
(* Moltiplica esclusivamente valori positivi.                        *)
(* ================================================================= *)
program Moltiplicare;

var     x       : integer;
```

```
        y       : integer;
        z       : integer;

(* ================================================================ *)
(* moltiplica( <x>, <y> )                                           *)
(* ---------------------------------------------------------------- *)
function moltiplica( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 0;

    for i := 1 to y do begin
        z := z+x;
    end;

    moltiplica := z;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := moltiplica( x, y );

    Write( x, ' * ', y, ' = ', z );

end.

(* ================================================================ *)
```

In alternativa si può tradurre il ciclo '**for**' in un ciclo '**while**':

```
function moltiplica( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 0;
    i := 1;

    while i <= y do begin
        z := z+x;
        i := i+1;
    end;

    moltiplica := z;

end;
```

Divisione intera tra due numeri positivi

«

Il problema della divisione tra due numeri positivi, attraverso la sottrazione, è descritto nella sezione 62.3.3.

```
(* ================================================================ *)
(* Dividi.pas                                                       *)
(* Divide esclusivamente valori positivi.                           *)
(* ================================================================ *)
program Dividere;

var     x       : integer;
        y       : integer;
        z       : integer;

(* ================================================================ *)
(* dividi( <x>, <y> )                                               *)
(* ---------------------------------------------------------------- *)
function dividi( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin
```

```
    z := 0;
    i := x;

    while i >= y do begin
        i := i - y;
        z := z+1;
    end;

    dividi := z;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := dividi( x, y );

    Write( x, ' / ', y, ' = ', z );

end.
(* ================================================================ *)
```

Elevamento a potenza

Il problema dell'elevamento a potenza tra due numeri positivi, attraverso la moltiplicazione, è descritto nella sezione 62.3.4.

```
(* ================================================================ *)
(* Exp.pas                                                          *)
(* Eleva a potenza.                                                 *)
(* ================================================================ *)
program Potenza;

var     x       : integer;
```

```
        y       : integer;
        z       : integer;

(* ================================================================= *)
(* exp( <x>, <y> )                                                   *)
(* ----------------------------------------------------------------- *)
function exp( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 1;

    for i := 1 to y do begin
        z := z * x;
    end;

    exp := z;

end;

(* ================================================================= *)
(* Inizio del programma.                                             *)
(* ----------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := exp( x, y );

    Write( x, ' ** ', y, ' = ', z );

end.
(* ================================================================= *)
```

In alternativa si può tradurre il ciclo '**`for`**' in un ciclo '**`while`**':

```
(* ================================================================= *)
(* exp( <x>, <y> )                                                   *)
(* ----------------------------------------------------------------- *)
function exp( x : integer; y : integer ) : integer;

var     z       : integer;
        i       : integer;

begin

    z := 1;
    i := 1;

    while i <= y do begin
        z := z * x;
        i := i+1;
    end;

    exp := z;

end;
```

È possibile usare anche un algoritmo ricorsivo:

```
function exp( x : integer; y : integer ) : integer;

begin

    if x = 0 then
        begin
            exp := 0;
        end
    else if y = 0 then
        begin
            exp := 1;
        end
    else
        begin
            exp := ( x * exp(x, y-1) );
        end
    ;

end;
```

«

Il problema della radice quadrata è descritto nella sezione 62.3.5.

```
(* ================================================================ *)
(* Radice.pas                                                       *)
(* Radice quadrata.                                                 *)
(* ================================================================ *)
program RadiceQuadrata;

var     x       : integer;
        z       : integer;

(* ================================================================ *)
(* radice( <x> )                                                    *)
(* ---------------------------------------------------------------- *)
function radice( x : integer; ) : integer;

var     z       : integer;
        t       : integer;
        ciclo   : boolean;

begin

    z := 0;
    t := 0;
    ciclo := TRUE;

    while ciclo do begin

        t := z * z;

        if t > x then
            begin
                z := z-1;
                radice := z;
                ciclo := FALSE;
            end
        ;

        z := z+1;

    end;
```

```
end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il numero intero positivo: ' );
    Readln( x );

    z := radice( x );

    Writeln( 'La radice di ', x, ' e'' ', z );

end.
(* ================================================================ *)
```

Fattoriale

«

Il problema del fattoriale è descritto nella sezione 62.3.6.

```
(* ================================================================ *)
(* Fact.pas                                                         *)
(* Fattoriale.                                                      *)
(* ================================================================ *)
program Fattoriale;

var     x       : integer;
        z       : integer;

(* ================================================================ *)
(* fact( <x> )                                                      *)
(* ---------------------------------------------------------------- *)
function fact( x : integer ) : integer;

var     i       : integer;

begin

    i := x - 1;

    while i > 0 do begin
```

```
        x := x * i;
        i := i-1;

    end;

    fact := x;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il numero intero positivo: ' );
    Readln( x );

    z := fact( x );

    Writeln( 'Il fattoriale di ', x, ' e'' ', z );

end.

(* ================================================================ *)
```

In alternativa, l'algoritmo si può tradurre in modo ricorsivo:

```
function fact( x : integer ) : integer;

begin

    if x > 1 then
        begin
            fact := ( x * fact( x - 1 ) )
        end
    else
        begin
            fact := 1
        end
    ;

end;
```

Massimo comune divisore

«

Il problema del massimo comune divisore, tra due numeri positivi, è descritto nella sezione 62.3.7.

```
(* ================================================================ *)
(* MCD.pas                                                          *)
(* Massimo Comune Divisore.                                         *)
(* ================================================================ *)
program MassimoComuneDivisore;

var     x       : integer;
        y       : integer;
        z       : integer;

(* ================================================================ *)
(* mcd( <x>, <y> )                                                  *)
(* ---------------------------------------------------------------- *)
function mcd( x : integer; y : integer ) : integer;

begin

    while x <> y do begin

        if x > y then
            begin
                x := x - y;
            end
```

```
        else
            begin
                y := y - x;
            end
        ;

    end;

    mcd := x;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il primo numero intero positivo: ' );
    Readln( x );
    Write( 'Inserisci il secondo numero intero positivo: ' );
    Readln( y );

    z := mcd( x, y );

    Write( 'Il massimo comune divisore tra ', x, ' e ', y, ' e'' ', z );

end.

(* ================================================================ *)
```

Numero primo

«

Il problema della determinazione se un numero sia primo o meno, è descritto nella sezione 62.3.8.

```
(* ================================================================ *)
(* Primo.pas                                                        *)
(* ================================================================ *)
program NumeroPrimo;

var     x       : integer;
```

```
(* ================================================================= *)
(* primo( <x> )                                                      *)
(* ----------------------------------------------------------------- *)
function primo( x : integer ) : boolean;

var     np      : boolean;
        i       : integer;
        j       : integer;

begin

    np := TRUE;
    i := 2;

    while (i < x) AND np do begin

        j := x / i;
        j := x - (j * i);

        if j = 0  then
            begin
                np := FALSE;
            end
        else
            begin
                i := i+1;
            end
        ;

    end;

    primo := np;

end;

(* ================================================================= *)
(* Inizio del programma.                                             *)
(* ----------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci un numero intero positivo: ' );
    Readln( x );
```

```
    if primo( x ) then
        begin
            Writeln( 'E'' un numero primo' );
        end
    else
        begin
            Writeln( 'Non e'' un numero primo' );
        end
    ;

end.
(* ================================================================ *)
```

Scansione di array

«

In questa sezione vengono mostrati alcuni algoritmi, legati alla scansione degli array, portati in Pascal.

Per semplicità, gli esempi mostrati fanno uso di array dichiarati globalmente, che come tali sono accessibili alle procedure e alle funzioni senza necessità di farne riferimento all'interno delle chiamate.

Ricerca sequenziale

«

Il problema della ricerca sequenziale all'interno di un array, è descritto nella sezione 62.4.1.

```
(* ================================================================ *)
(* RicercaSeq.pas                                                   *)
(* Ricerca sequenziale.                                             *)
(* ================================================================ *)
program RicercaSequenziale;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        x       : integer;
```

```
        i       : integer;
        z       : integer;

(* ================================================================ *)
(* ricercaseq( <x>, <ele-inf>, <ele-sup> )                          *)
(* ---------------------------------------------------------------- *)
function ricercaseq( x : integer; a : integer; z : integer ) : integer;

var     i       : integer;

begin

    (* ------------------------------------------------------------ *)
    (* Se l'elemento non viene trovato, il valore -1 segnala        *)
    (* l'errore.                                                    *)
    (* ------------------------------------------------------------ *)
    ricercaseq := -1;

    (* ------------------------------------------------------------ *)
    (* Scandisce l'array alla ricerca dell'elemento.                *)
    (* ------------------------------------------------------------ *)
    for i := a to z do begin

        if x = lista[i] then
            begin
                ricercaseq := i;
            end
        ;

    end;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
```

```
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    Writeln( 'Inserire il valore da cercare' );
    Readln( x );

    i := ricercaseq( x, 1, z );

    Writeln( 'Il valore cercato si trova nell''elemento', i );

end.
(* ================================================================ *)
```

Esiste anche una soluzione ricorsiva che viene mostrata nella subroutine seguente:

```
function ricercaseq( x : integer; a : integer; z : integer ) : integer;

begin

    if a > z then
        begin

            (* ------------------------------------------------- *)
            (* La corrispondenza non è stata trovata.            *)
            (* ------------------------------------------------- *)
            ricercaseq := -1;
        end
    else if x = lista[a] then
        begin
            ricercaseq := a;
        end
    else
        begin
            ricercaseq := ricercaseq( x, a+1, z);
        end
    ;

end;
```

Ricerca binaria

«

Il problema della ricerca binaria all'interno di un array, è descritto nella sezione 62.4.2.

```
(* ================================================================ *)
(* RicercaBin.pas                                                   *)
(* Ricerca binaria.                                                 *)
(* ================================================================ *)
program RicercaBinaria;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        x       : integer;
        i       : integer;
        z       : integer;

(* ================================================================ *)
(* ricercabin( <x>, <ele-inf>, <ele-sup> )                          *)
```

```
(* ------------------------------------------------------------- *)
function ricercabin( x : integer; a : integer; z : integer ) : integer;

var     m       : integer;

begin

    (* ------------------------------------------------------------- *)
    (* Determina l'elemento centrale.                                *)
    (* ------------------------------------------------------------- *)
    m := ( a + z ) / 2;

    if m < a then
        begin

            (* ------------------------------------------------------ *)
            (* Non restano elementi da controllare.                   *)
            (* ------------------------------------------------------ *)
            ricercabin := -1;
        end
    else if x < lista[m] then
        begin

            (* ------------------------------------------------------ *)
            (* Si ripete la ricerca nella parte inferiore.            *)
            (* ------------------------------------------------------ *)
            ricercabin := ricercabin( x, a, m-1 );
        end
    else if x > lista[m] then
        begin

            (* ------------------------------------------------------ *)
            (* Si ripete la ricerca nella parte superiore.            *)
            (* ------------------------------------------------------ *)
            ricercabin := ricercabin( x, m+1, z );
        end
    else
        begin

            (* ------------------------------------------------------ *)
            (* m rappresenta l'indice dell'elemento cercato.          *)
            (* ------------------------------------------------------ *)
            ricercabin := m;
        end
```

```
    ;

end;

(* ================================================================= *)
(* Inizio del programma.                                             *)
(* ----------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    Writeln( 'Inserire il valore da cercare' );
    Readln( x );

    i := ricercabin( x, 1, z );

    Writeln( 'Il valore cercato si trova nell''elemento', i );

end.
(* ================================================================= *)
```

Algoritmi tradizionali

In questa sezione vengono mostrati alcuni algoritmi tradizionali portati in Pascal.

«

Il problema del Bubblesort è stato descritto nella sezione 62.5.1. Viene mostrata prima una soluzione iterativa e successivamente la funzione '**bsort**' in versione ricorsiva.

```
(* ================================================================ *)
(* BSort.pas                                                        *)
(* ================================================================ *)
program BubbleSort;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        i       : integer;
        z       : integer;

(* ================================================================ *)
(* bsort( <ele-inf>, <ele-sup> )                                    *)
(* ---------------------------------------------------------------- *)
procedure bsort( a : integer; z : integer );

var     scambio : integer;
        j       : integer;
        k       : integer;

begin

    (* ------------------------------------------------------------ *)
    (* Inizia il ciclo di scansione dell'array.                     *)
    (* ------------------------------------------------------------ *)
    for j := a to ( z-1 ) do begin

        (* -------------------------------------------------------- *)
        (* Scansione interna dell'array per collocare nella         *)
        (* posizione j l'elemento giusto.                           *)
        (* -------------------------------------------------------- *)
        for k := ( j+1 ) to z do begin

            if lista[k] < lista[j] then
                begin

                    (* ------------------------------------------- *)
                    (* Scambia i valori.                           *)
```

```
                    (* ------------------------------------------ *)
                    scambio := lista[k];
                    lista[k] := lista[j];
                    lista[j] := scambio;
                end
            ;

        end;
    end;

end;

(* ================================================================= *)
(* Inizio del programma.                                             *)
(* ----------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    bsort( 1, z );

    Writeln( 'Array ordinato:' );

    for i := 1 to z do begin
        Write( lista[i] );
    end;

end.
```

```
(* ============================================================== *)
```

Segue la procedura '**bsort**' in versione ricorsiva:

```
procedure bsort( a : integer; z : integer );

var     scambio : integer;
        k       : integer;

begin

    if a < z then
        begin

            (* ------------------------------------------------- *)
            (* Scansione interna dell'array per collocare nella  *)
            (* posizione j l'elemento giusto.                    *)
            (* ------------------------------------------------- *)
            for k := ( a+1 ) to z do begin

                if lista[k] < lista[a] then
                    begin

                        (* ------------------------------------- *)
                        (* Scambia i valori.                     *)
                        (* ------------------------------------- *)
                        scambio := lista[k];
                        lista[k] := lista[a];
                        lista[a] := scambio;
                    end
                ;

            end;

            bsort( a+1, z );

        end
    ;

end;
```

Il problema della torre di Hanoi è descritto nella sezione 62.5.3.

```
(* ================================================================ *)
(* Hanoi.pas                                                        *)
(* Torre di Hanoi.                                                  *)
(* ================================================================ *)
program TorreHanoi;

var     n       : integer;
        p1      : integer;
        p2      : integer;

(* ================================================================ *)
(* hanoi( <n>, <p1>, <p2> )                                         *)
(* ---------------------------------------------------------------- *)
procedure hanoi( n : integer; p1 : integer; p2 : integer );

begin

    if n > 0 then
        begin
            hanoi( n-1, p1, 6-p1-p2 );

            Writeln(
                'Muovi l''anello ', n:1,
                ' dal piolo ', p1:1,
                ' al piolo ', p2:1
            );

            hanoi( n-1, 6-p1-p2, p2 );
        end
    ;

end;

(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln;
    Write( 'Inserisci il numero di anelli: ' );
```

```
    Readln( n );
    Write( 'Inserisci il piolo iniziale: ' );
    Readln( p1 );
    Write( 'Inserisci il piolo finale: ' );
    Readln( p2 );

    hanoi( n, p1, p2 );

end.
(* ================================================================= *)
```

Quicksort

«

L'algoritmo del Quicksort è stato descritto nella sezione 62.5.4.

```
(* ================================================================= *)
(* QSort.pas                                                         *)
(* ================================================================= *)
program QuickSort;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        i       : integer;
        z       : integer;

(* ================================================================= *)
(* part( <ele-inf>, <ele-sup> )                                      *)
(* ----------------------------------------------------------------- *)
function part( a : integer; z : integer ) : integer;

var     scambio : integer;
        i       : integer;
        cf      : integer;
        loop1   : boolean;
        loop2   : boolean;
        loop3   : boolean;

begin

    (* ------------------------------------------------------------- *)
    (* Si assume che a sia inferiore a z.                            *)
    (* ------------------------------------------------------------- *)
```

```
    i := a+1;
    cf := z;

    (* ------------------------------------------------------------ *)
    (* Inizia il ciclo di scansione dell'array.                     *)
    (* ------------------------------------------------------------ *)
    loop1 := TRUE;
    while loop1 do begin

        loop2 := TRUE;
        while loop2 do begin

            (* ------------------------------------------------- *)
            (* Sposta i a destra.                                *)
            (* ------------------------------------------------- *)
            if ( lista[i] > lista[a] ) OR ( i >= cf ) then
                begin
                    loop2 := FALSE;
                end
            else
                begin
                    i := i+1;
                end
            ;

        end;

        loop3 := TRUE;
        while loop3 do begin

            (* ------------------------------------------------- *)
            (* Sposta cf a sinistra.                             *)
            (* ------------------------------------------------- *)
            if lista[cf] <= lista[a] then
                begin
                    loop3 := FALSE;
                end
            else
                begin
                    cf := cf-1;
                end
            ;

        end;
```

```
        if cf <= i then
            begin

                (* ------------------------------------------------ *)
                (* è avvenuto l'incontro tra i e cf.                *)
                (* ------------------------------------------------ *)
                loop1 := FALSE;
            end
        else
            begin

                (* ------------------------------------------------ *)
                (* Vengono scambiati i valori.                      *)
                (* ------------------------------------------------ *)
                scambio := lista[cf];
                lista[cf] := lista[i];
                lista[i] := scambio;

                i := i+1;
                cf := cf-1;
            end
        ;
    end;

    (* ------------------------------------------------------------ *)
    (* A questo punto, lista[a..z] è stata ripartita e cf è la      *)
    (* collocazione finale.                                         *)
    (* ------------------------------------------------------------ *)
    scambio := lista[cf];
    lista[cf] := lista[a];
    lista[a] := scambio;

    (* ------------------------------------------------------------ *)
    (* In questo momento, lista[cf] è un elemento (un valore) nella *)
    (* posizione giusta.                                            *)
    (* ------------------------------------------------------------ *)
    part := cf

end;

(* ================================================================ *)
(* quicksort( <ele-inf>, <ele-sup> )                                *)
(* ---------------------------------------------------------------- *)
```

```
procedure quicksort( a : integer; z : integer );

var     cf      : integer;

begin

    if z > a then
        begin
            cf := part( a, z );
            quicksort( a, cf-1 );
            quicksort( cf+1, z );
        end
    ;

end;

(* ================================================================= *)
(* Inizio del programma.                                             *)
(* ----------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    quicksort( 1, z );

    Writeln( 'Array ordinato:' );

    for i := 1 to z do begin
        Write( lista[i] );
```

```
    end;

end.
(* ============================================================== *)
```

Permutazioni

«

L'algoritmo ricorsivo delle permutazioni è descritto nella sezione 62.5.5.

```
(* ============================================================== *)
(* Permuta.pas                                                    *)
(* ============================================================== *)
program Permutazioni;

const   DIM     = 100;

var     lista   : array[1..DIM] of integer;
        i       : integer;
        z       : integer;

(* ============================================================== *)
(* permuta( <ele-inf>, <ele-sup>, <elementi-totali> )             *)
(* -------------------------------------------------------------- *)
function permuta( a : integer; z : integer; elementi : integer ) : integer;

var     scambio : integer;
        k       : integer;
        i       : integer;

begin

    (* ---------------------------------------------------------- *)
    (* Se il segmento di array contiene almeno due elementi,      *)
    (* si procede.                                                *)
    (* ---------------------------------------------------------- *)
    if ( z-a ) >= 1 then
        begin

            (* ------------------------------------------------- *)
            (* Inizia il ciclo di scambi tra l'ultimo elemento e  *)
            (* uno degli altri contenuti nel segmento di array.   *)
```

```
            (* ------------------------------------------------- *)
            k := z;
            while k >= a do begin

                (* ------------------------------------------------- *)
                (* Scambia i valori.                                 *)
                (* ------------------------------------------------- *)
                scambio := lista[k];
                lista[k] := lista[z];
                lista[z] := scambio;

                (* ------------------------------------------------- *)
                (* Esegue una chiamata ricorsiva per permutare un    *)
                (* segmento più piccolo dell'array.
                (* ------------------------------------------------- *)
                permuta( a, z-1, elementi );

                (* ------------------------------------------------- *)
                (* Scambia i valori.                                 *)
                (* ------------------------------------------------- *)
                scambio := lista[k];
                lista[k] := lista[z];
                lista[z] := scambio;

                k := k-1;

            end;
        end
    else
        begin

            (* ------------------------------------------------- *)
            (* Visualizza la situazione attuale dell'array.      *)
            (* ------------------------------------------------- *)
            for i := 1 to elementi do begin
                Write( lista[i]:4 );
            end;
            Writeln;

        end
    ;

end;
```

```
(* ================================================================ *)
(* Inizio del programma.                                            *)
(* ---------------------------------------------------------------- *)
begin

    Writeln( 'Inserire il numero di elementi.' );
    Writeln( DIM, ' al massimo.' );
    Readln( z );

    if z > DIM then
        begin
            z := DIM;
        end
    ;

    Writeln( 'Inserire i valori dell''array' );

    for i := 1 to z do begin
        Write( 'elemento ', i:2, ': ' );
        Readln( lista[i] );
    end;

    permuta( 1, z, z );

end.

(* ================================================================ *)
```