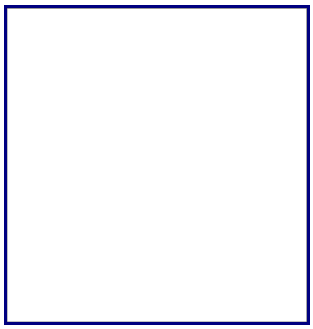
Benvenuto nel wikibook:

Pascal



Edizione: 0.2 4/06/07

Versione attuale del libro>>

Autori: Flash, Mmo, Pietrodn, Ramac, Xno,.

Copertina: Blaise Pascal, ed è rilasciata secondo i termini della GFDL.

Indice generale

Premessa....2
Finalità....3
Introduzione....3
Hello world!....3
Primi concetti sulla sintassi....4

Le variabili....4
Assegnazione....5
Altri concetti di sintassi....5
Tipi di dati....6
Integer....6
Char....7
Real....7
Boolean....8
String....8
Tabella: Operazioni ed operatori....9
Type....11
Commenti....11
Input e Output....12
Read e Readln....13
Write e Writeln....13
Formattare i numeri....13
Librerie e funzioni predefinite....14
crt....15
Funzioni di crt....15
Tabella: Le frequenze delle note....16
graph....17
Funzioni di graph....17
graph3....19
Funzioni di graph3....19
Istruzioni di controllo....20
If, then, else....20
Case...of, else....21
Strutture Iterative....21
For....21
While do....23
Repeat Until....24
For, While, Repeat Until....24
Array....24
Vettori....24
Gli array in Pascal....25
Array bidimensionali....26
Metodo top-down....26
Procedure....27
Procedure con argomenti....28
Passaggio di parametri per valore e per riferimento....29
Funzioni....30
Strumenti....30
Compilatori....30
Problemi di compatibilità....31
IDE....31
Alcuni esempi commentati....32
Licenza....34

Premessa

Il Pascal è un linguaggio di programmazione caratterizzato da una estrema rigidità basato sul

concetto di programmazione strutturata.
Le sue peculiarità sono date dall'intenzione del suo creatore, Niklaus Wirth, di creare un linguaggio che, a differenza del BASIC, forzasse da parte del programmatore uno studio accurato dell'algoritmo e che fosse dotato di funzionalità e strutture dati avanzate.
Questi elementi lo rendono uno dei linguaggio più usati a livello didattico, soprattutto nelle scuole.

Finalità

Il libro non richiede particolari conoscenze di programmazione per essere letto; inoltre porta il lettore ad una discreta conoscenza del linguaggio Pascal a livelli comunque abbastanza semplici.

Introduzione

Il **Pascal** è un linguaggio di programmazione nato nel 1973 ad opera di *Niklaus Wirth*.
Egli, oltre ad essere un informatico, era anche un insegnante di programmazione: creò il Pascal proprio come linguaggio didattico e lo progettò appositamente molto rigido; questo perché richiedesse prima dell'implementazione un'attenta analisi da parte dell'alunno dell'algoritmo da sviluppare.
Il Pascal è inoltre un linguaggio piuttosto facile da usare, anche per la sua somiglianza con la lingua inglese che rende alcuni comandi piuttosto intuitivi.

Ancora oggi, il Pascal è il linguaggio più usato a livello didattico; per questo motivi è utilizzato diffusamente come linguaggio introduttivo alla programmazione strutturata. Questo però non significa però che abbia poche potenzialità. Wirth pensava al Pascal come un linguaggio facile ma potente; addirittura la sua evoluzione orientata agli oggetti, il Delphi, è estremamente diffusa in vari ambiti.

Hello world!

Storicamente, ogni volta che si deve presentare un linguaggio di programmazione, si utilizza il mitico *Hello World*, un piccolo programma che stampa sullo schermo, appunto, *Hello World!*. In Pascal la sua realizzazione è piuttosto semplice:

```
program HelloWorld;
begin
     writeln('Hello world!');
end.
```

- la prima riga indica il nome del programma. La parola *program* è una parola riservata del linguaggio. Non può quindi essere usata come nome di variabile o di funzione (che vedremo più avanti nel corso del libro). Alla fine di ogni istruzione bisogna inserire il carattere *;* per indicare al compilatore che il comando è concluso.
- la seconda riga indica l'inizio del programma vero e proprio. Ogni blocco di codice deve essere inserito fra le parole riservate `begin` e `end`.
- la terza riga contiene la nostra prima vera istruzione. I comandi di input-output verranno trattati in seguito. Per ora basti sapere che l'istruzione `writeln` permette di mostrare sullo schermo i caratteri inseriti fra gli apici.
- l'ultima riga rappresenta la conclusione del programma. La parola riservata *end* posta alla fine del programma deve essere sempre seguita dal punto.

Per alcuni compilatori (soprattutto in ambiente Windows) sarà necessario aggiungere un'ultima riga di codice, `readln;`, prima di `end`. L'istruzione infatti blocca l'esecuzione del programma fino a che questi non preme <Enter>, permettendo così all'utente di vedere l'output del programma prima che si chiuda la finestra in cui è stato fatto partire. L'istruzione `readln;` verrà comunque trattata approfonditamente più avanti nel corso di questo libro.

Primi concetti sulla sintassi

Dal nostro primo esempio possiamo ricavare alcune semplici indicazioni sulla sintassi di un programma implementato in Pascal:

- esistono delle **parole riservate**, che fanno parte della grammatica del linguaggio Pascal, che possono essere usate solo con lo scopo prefissato.
- ogni programma è composto da **istruzioni**, semplici o composte.
- le istruzioni vengono eseguite dalla prima all'ultima in **sequenza**.
- le istruzioni composte vengono detti **blocchi**. Vengono individuati da una coppia di parole chiave: *begin* indica l'inizio del blocco, *end* la fine e contengono a loro volta istruzioni.
- il blocco può essere considerato come un'unica istruzione.
- ogni istruzione è separata da un carattere speciale: *;* il punto e virgola.
- esistono degli identificatori, ovvero nomi arbitrari per variabili, costanti, procedure e funzioni (che vedremo più avanti).

Le variabili

Un programma come quello appena fatto può essere divertente e interessante, ma offre ben poche possibilità di sviluppo. Quasi tutti i tipi di programmi richiedono infatti l'uso di calcoli e di "valori da ricordare". Per questo motivo è necessario introdurre il concetto di **variabile**.

Una variabile può essere paragonata ad una casella di una certa dimensione nella quale si possono inserire e/o leggere dati mentre il programma è in esecuzione. In pratica le variabili sono delle etichette che noi diamo ad un'area di memoria, che verrà usata per depositare dei dati. La quantita di memoria riservata dipende dal tipo di variabile che andiamo a dichiarare e dal particolare compilatore che usiamo.

Il compilatore Pascal ha la necessita di conoscere l'uso e lo scopo di tutte le etichette che incontra durante la fase di compilazione, è indispensabile quindi dichiarare esplicitamente, in particolari punti del programma, le variabili e altro che vedremo. L'area di dichiarazione delle variabili inizia con la parola chiave *var*.
Il concetto risulta più chiaro con un esempio:

```
program VariabiliVarie;
var
    n:integer;
    r:real;
begin
     n:=3;
     r:=sqrt(n);
     n:=5;
     writeln(n);
     writeln(r);
end.
```

Il programma in sè è estremamente sciocco, ma ci permette di osservare come le variabili vengano utilizzate in Pascal. Analizzamo le singole righe:

- la prima riga è l'intestazione del programma (alla parola riservata *program* segue il nome scelto per il programma)
- nella seconda e nella terza riga vengono dichiarate le variabili. La dichiarazione ha sempre la forma

```
var
    nomevariabile : tipo;
    nomevariabile2 : tipo2;
```

> Il Pascal non permette di dichiarare durante il programma variabili aggiuntive, perciò è necessaria una buona progettazione teorica del programma per non trovarsi a dover correggere molti errori in corso di compilazione (ovviamente ciò era tra le intenzioni di Wirth quando progettò il linguaggio).

Assegnazione

Una delle operazioni fondamentali che si possono eseguire con le variabile è l'istruzione di assegnazione.
Per assegnare si intende attribuire ad una determinata variabile un valore specificato dal programmatore oppure il risultato di un'espressione (per vedere come operare con le variabili, leggi il prossimo modulo)

La **sintassi** dell'assegnazione è piuttosto semplice:

```
nome_della_variabile := valore_assegnato_alla_variabile;
```

dove valore_assegnato_alla_variabile è un valore o un'espressione il cui risultato è del tipo di dato dichiarato all'inizio del programma. Se infatti si tenta di assegnare un valore del tipo *x* ad una variabile del tipo *y* il compilatore restituisce un errore.

Altri concetti di sintassi

Abbiamo visto due esempi. Possiamo notare che entrambi riflettono anche uno stile di scrittura del codice, che diciamo **stile di formattazione**. Notiamo che il codice presenta varie rientranze. Queste hanno lo scopo di rendere più semplice la vita del programmatore. Consentono di individuare a colpo d'occhio un blocco, semplicemente osservando l'andamento delle rientranze.
Per il compilatore tutto ciò è ininfluente, in quanto lui individua i blocchi tramite le parole chiavi e trova le istruzioni separate da un carattere apposito (il punto e virgola).

Se provate a compilare le seguenti righe otterrete lo stesso programma visto nell'esempio precedente. Per il compilatore non ci sono differenze, mentre per il programmatore che dovrà estendere o correggere questo codice le cose si complicano.

```
program VariabiliVarie;
var n:integer; r:real;
begin
     n:=3;
     r:=sqrt(n);
     n:=5;
     writeln(n);
```

```
     writeln(r);
end.
```

Per lo stesso motivo anche un programma come questo è valido:

```
program VariabiliVarie;  var n:integer; r:real; begin n:=3; r:=sqrt(n); n:=5;
writeln(n); writeln(r); end.
```

ma è ovviamente molto meno leggibile.

Ma quali sono i tipi di variabile utilizzabili?

Tipi di dati

Integer

Il tipo più utilizzato è in genere il tipo *integer*. Corrisponde agli interi, anche se non proprio a tutti;comprende gli interi da -32768 a 32767,e occupa una memoria di 16 bit(2 byte). Essendo il computer una macchina limitata, non è possibile fare somme fino all'infinito. Infatti con alcuni semplici programmi si può osservare che esiste un limite superiore per gli interi (come nel caso del tipo *integer*,cioè 32767), oltre al quale si ricade nel limite inferiore, come se fosse una circonferenza. Il fenomeno dello sforamento del limite massimo è detto Overflow.

Ogni tipo di dato ha delle operazioni permesse. In questo caso le operazioni che danno come risultato un integer sono:

- + permette di sommare due valori di tipo integer.
- - permette di calcolare la differenza fra due valori integer.
- * permette di moltiplicare due valori, ottenendo sempre un integer.
- *div* permtte di calcolare la parte intera del rapporto fra due interi.
- *mod* permette di calcolare il resto della divisione fra due interi.

Le funzioni matematiche che danno come risultato un intero sono, invece

- *abs(n)* che calcola il valore assoluto del numero intero *n*
- *round(n)* che arrotonda qualunque numero all'intero più vicino ad *n*
- *trunc(n)* che tronca il numero all'intero minore di *n*
- *sqr(n)* ne calcola infine il quadrato.

Proviamo a fare un programma che utilizzi alcune operazioni. Per ora ci limiteremo a fare semplici assegnamenti di variabili all'interno del programma stesso. Presto troveremo il modo di dinamicizzare le cose...

```
program Pari;
var n:integer;
begin
     n:=5;
     if (n mod 2 = 0) then write ('pari')
                      else write ('dispari');
end.
```

Il programma presenta alcuni costrutti che non abbiamo ancora visto, ma la cui intepretazione è

semplice. Analizziamo come al solito riga per riga.

- Come al solito la prima riga contiene la dichiarazione del titolo.
- Dichiarazione di una sola variabile *n* di tipo intero.
- Inizia il programma.
- Assegnamo a *n* il valore 5.
- Inizia un costrutto condizionale. Letto "alla lettera" significa "se il resto della divisione fra n e 2 è uguale a zero, allora scrivi *pari* altrimenti scrivi *dispari*". Da notare, lo rivedremo e faremo alcune considerazioni, che prima di *else* non va **mai** messo il ";".

Char

Frequentemente si utilizzano invece degli *integer* i *char*, per rappresentare numeri interi piccoli. Infatti i *char* rappresentano il range di numeri interi che va da 0 a 255 - corrispondente ai codici ASCII - e permettono la visualizzazione nei due formati: intero, ascii. Le operazioni possibili sono le stesse degli integer, ma ci sono due funzioni in più che permettono la trasformazione da intero a carattere e viceversa. Queste funzioni sono

- *chr(x)* che permette di trasformare l'intero *x* nel corrispondente carattere
- *ord(x)* che permette di trasformare il carattere *x* nel corrispondente intero ASCII

Un semplice esempio di uso potrebbe essere la stampa della tabella dei codici ascii.

```
program ASCII;
var n:integer;
begin
     for n:=0 to 255 do
     writeln(n, ' ==> ', chr(n));
end.
```

Come al solito analizziamo riga per riga.

- Dichiarazione del programma,
- Definizione della variabile *n* come char
- Inizio del programma
- Il costrutto *for...to...do* non l'abbiamo ancora visto, ma facendo come prima la traduzione letterale, possiamo intuirne il significato: *Per n che va da 0 a 255 fai...* Questo tipo di costrutto, detto *ciclo*, verrà comunque affrontato in seguito.
- scriviamo su schermo il numero *n* seguito dai caratteri ==> e poi dal corrispondente carattere ASCII.

Per assegnare ad una variabile *char* un valore è necessario usare la notazione

```
nome_variabile := 'carattere';
```

I due caratteri ' (carattere apice) identificano un valore alfanumerico e non numerico.

Real

Le variabili di tipo real corrispondono ai numeri reali, ovvero i numeri con la virgola. Anche qui, come per gli integer, dobbiamo dare delle limitazioni. Essendo, come già sottolineato un computer

una macchina limitata e discreta, inevitabilmente non può superare una certa precisione nel calcolo con i numeri reali. Per questo motivo spesso nelle strutture condizionate si devono cercare di utilizzare stratagemmi per evitare problemi.

Le operazioni di base possibili sembrano meno di quelle possibili con char e integer, ma attraverso l'utilizzo della libreria matematica aumentano in modo incredibile. Comunque le operazioni basilari sono:

- + permette di sommare due valori di tipo real.
- - permette di calcolare la differenza fra due valori real.
- * permette di moltiplicare due valori, ottenendo sempre un real.
- / permette di calcolare il rapporto fra due real.

Interne all'insieme dei reali troviamo ad esempio funzioni come:

- *sin(a)* e *cos(a)*: restituiscono il seno e il coseno dell'angolo a (espresso in radianti)
- *random*: fornisce un numero casuale compreso tra 0 e 1 esclusi. Prima di usare questa funzione è bene usaare *randomize*, che reinizializza il generatore di numeri casuali.

Boolean

L'algebra booleana è fondamentale nell'informatica. Questa permette infatti di fare calcoli sulla veridicità o falsità di una affermazione. Le variabili booleane infatti possono assumere solo i valori logici vero (*true*) e falso (*false*). Le operazioni possibili sono diverse da quelle possibili per gli altri tipi di variabile:

- *and* che corrisponde al simbolo matematico $\wedge$ e al concetto di *e contemporaneamente*
- *or* che rappresenta $\vee$ e *oppure*
- *not* operatore che corrisponde alla negazione
- *xor* che corrisponde matematicamente all'*aut*, ovvero $\dot{\vee}$

Le variabili di tipo booleano si dichiarano con la parola riservata *boolean*.

```
var
 variabile_booleana:boolean;
```

Si inizializzano così:

```
variabile_booleana:=true;
 oppure
 variabile_booleana:=false;
```

L'utilizzo di variabili Boolean è in genere limitato all'analisi di particolari aspetti dell'input da utente, o all'utilizzo come 'flag' nella programmazione più avanzata.

String

Le variabili string sono variabili che possono contenere una stringa alfanumerica di caratteri.

La dichiarazione di una variabile string ha sintassi:

```
nome_della_variabile : string[n];
```

Ovvero afferma che la variabile *nome_della_variabile* può contenere una riga di massimo *n* caratteri.

Come per le variabili *char*, anche le variabili *string* necessitano una sintassi particolare per l'assegnazione: l'istruzione

```
nome_variabile := 'Ma la volpe col suo balzo ha raggiunto il quieto fido 1234';
```

assegna a nome variabile la stringa alfanumerica (in questo caso *Ma la volpe col suo balzo ha raggiunto il quieto fido 1234*.

Nel caso si voglia inserire un carattere apice nella propria stringa (per inserire ad esempio un apostrofo) è necessario usare due apici di seguito. Un'istruzione come questa

```
writeln('L'unica soluzione...');
```

genera infatti un errore, poiché l'apostrofo tra *La* e *unica* è interpretato dal compilatore come un apice che delimita la fine della stringa. Un'istruzione invece come questa

```
writeln('L''unica soluzione...');
```

è interpretata correttamente.

L'unico operatore possibile con il tipo string è l'operatore di concatenazione +, che è possibile capire con un esempio:

```
var1 := 'Ma la volpe col suo balzo ';
var2 := 'ha raggiunto il quieto fido';
var3 := var1 + var2 + ' 1234';
```

La variabile var3 alla fine di questo breve listato conterrà così il valore *Ma la volpe col suo balzo ha raggiunto il quieto fido 1234*, in quanto le stringhe sono state concatenate in una stringa unica. Si noti lo spazio inserito prima dell'apice: senza lo spazio, le parole risulterebbero attaccate.

Una funzionalità interessante relativa alle stringhe è l'**indicizzazione**, per la quale è possibile fare riferimento ad un carattere all'interno della stringa, usando la notazione

```
variabile[n] {l'n-esimo carattere della stringa in variabile}
```

Tabella: Operazioni ed operatori

Operatore	Operandi	Risultato
+	real o integer o char	real o integer o char
-	real o integer o char	real o integer o char
*	real o integer o char	real o integer o char
/	real o integer o	real

	char	
mod	integer o char	integer
div	integer o char	integer
Funzione	**Argomento**	**Risultato**
sqr(*x*)	real o integer o char	integer
sqrt(*x*)	real o integer o char	real
abs(*x*)	real o integer o char	real o integer o char
trunc(*x*)	real o integer o char	integer
sin(*x*) e cos(*x*)	real o integer o char	real
random	-	real

Il significato di questi operatori e di queste funzioni è stato mostrato nei paragrafi precedenti. ad eccezione della funzione *sqrt(*n*)*, che restituisce la radice quadrata di un *n*.

Pascal mette a disposizione inoltre alcuni operatori di confronto che restituiscono sempre un valore boolean e che possono essere usati su due variabili qualsiasi dello stesso tipo di dato:

- < (minore)
- <= (minore o uguale)
- = (uguale)
- <> (diverso) - si può usare anche not(val1 = val2)
- >= (maggiore o uguale)
- > (maggiore)

È intuibile il loro funzionamento nell'ambito di variabili integer o real; questi operatori tuttavia possono essere anche usati sui tipi di dato char o string; più in particolare:

```
char1 < char2
```

è equivalente a scrivere

```
ord(char1) < ord(char2)
```

La comparazione tra valori string valuta invece le stringhe in ordine alfabetico (quindi saranno vere le espressioni come, ad esempio, 'abaco' < 'amico' o 'zaino' = 'zaino').
Per quanto riguarda i valori boolean, vale

```
false < true
```

e, quindi

```
true > false
```

Type

Abbiamo già visto prima che per dichiarare una variabile si deve specificare un tipo (ad esempio, integer o real) per indicare quali valori può assumere. In alcuni casi può essere utile ricorrere a delle tipologie di dati non predefinite: si può ricorrere a una *dichiarazione di tipo* con la parola riservata **type**.

La sintassi è

```
type
 nome:(elemento_1, elemento_2,...,elemento_n);
```

dove elemento_1,elemento_2,...,elemento_n sono i possibile valori che possono assumere le variabili del tipo di dato *nome* elencati in ordine.

Per indicare quindi che una variabile è del tipo *nome* si usa la sintassi:

```
var
 variabile:nome;
```

Per fare un esempio, possiamo definire un tipo:

```
type
 mesi: (gen, feb, mar, apr, mag, giu, lug, ago, set, ott, nov, dic);
```

e dichiarare due variabili

```
var
 mese_di_nascita_luigi, mese_di_nascita_gianni:mesi;
```

Se corso del programma poniamo

```
mese_di_nascita_luigi:= feb;
mese_di_nascita_gianni:= ago;
```

avremo che: l'istruzione

```
writeln(mese_di_nascita_luigi);
```

stampa sullo schermo *feb*; l'espressione

```
mese_di_nascita_luigi < mese_di_nascita_gianni
```

restituisce TRUE, in quanto feb precede ago nella lista dei mesi indicata nella dichiarazione. Allo stesso modo

```
giu > set
```

restituisce invece FALSE.

Commenti

In programmazione, un **commento** è una parte del listato che viene ignorata dal compilatore, ma che è funzionale a chi ha steso il codice al fine di lasciare qualche appunto, specie se il codice in

questione deve essere letto da altri programmatori.

In Pascal i commenti si indicano delimitandoli da parentesi graffe (digitabili tramite Shift+Alt Gr+tasto delle parentesi quadre), o da parentesi tonde seguite e precedute da un asterisco; ad esempio:

```
{Questo è un commento}
```

oppure

```
(*Questo è un commento*)
```

L'utilità dei commenti è evidente nei casi di scambio dei sorgenti di un programma all'interno di una comunità di sviluppatori: è spesso difficile riuscire infatti a capire il funzionamento di un algoritmo studiato e codificato da qualcun altro senza l'ausilio di commenti scritti in maniera efficace.
I commenti possono rivelarsi utili inoltre nel caso un autore di un programma riprenda il suo lavoro dopo un periodo di pausa: nel caso il codice non fosse ben commentato, infatti, l'autore si ritroverebbe nella stessa situazione di un qualsiasi programmatore che non ha mai letto il codice e gli sarebbe quindi difficile riprendere il lavoro in maniera adeguata.

Input e Output

Finalmente ci occupiamo del grande problema dell'**input-output**.
Gran parte delle funzioni di input-output sono contenute nella libreria standard di Pascal. Queste funzioni sono tipicamente:

- *read()* e *readln()* che permettono di leggere l'input dell'utente da tastiera inserendo fra le parentesi il nome della variabile in cui vogliamo salvare il dato. La differenza consiste nel cursore, che nel caso di un'istruzione *read* continua sulla stessa riga mentre nel caso di *readln* va a capo.
- *write()* e *writeln()* che permettono, come già visto in precedenza, di stampare su schermo il contenuto delle parentesi. Anche in questo caso la differenza è relativa all'andare a capo

Un semplice esempio basato sull'uso delle variabili e dell'input-output potrebbe essere un modo per personalizzare un programma, salvando in una stringa il nome dell'utente per inserirlo nelle domande.

```
program username;
var name: String[50];
uses crt;
begin
     clrscr;
     writeln('Inserisci il nome');
     readln(name);
     clrscr;
     writeln('Benvenuto, ', name);
     readkey;
end.
```

Analizziamo come al solito riga per riga:

- dichiarazione del nome
- dichiarazione della variabile *name* di tipo Stringa di dimensione 50
- dichiarazione delle librerie necessarie. Di questo argomento riparleremo a breve, ma

sappiate che almeno due dei comandi utilizzati di seguito sono dipendenti da questa libreria
- inizio programma
- pulizia dello schermo. Questa è la prima (e forse la più usata) delle due funzioni che dipendono da *crt*.
- stampa sullo schermo della stringa 'Inserisci il nome'. Dopo questa istruzione il programma va a capo
- lettura della stringa corrispondente al nome ed inserimento di questa stringa letta da tastiera nella variabile *name*
- nuova pulizia dello schermo
- stampa del messaggio di benvenuto
- altra funzione dipendente dalla libreria *crt* che permette di leggere un solo carattere qualunque da tastiera. Vedremo che questa funzione è molto utilizzata per risolvere un problema riguardo all'esecuzione dei programmi compilati finora.

Read e Readln

Il contenuto delle parentesi che seguono l'istruzione *read* o *readln* può essere una o più variabili. Nel secondo caso, in fase di esecuzione, i diversi valori di input vanno inseriti separandoli con uno spazio.

Write e Writeln

Il contenuto della parentesi che seguono l'istruzione *writeln* può essere sia un'espressione (stringa o numerica), sia una variabile sia una combinazione.
La parte testuale è compresa tra due apici; per separare il testo dalle variabili si usa una virgola, come nell'esempio precedente.

Alcuni compilatori, soprattutto i più vecchi, possono inoltre dare problemi nella stampa dei caratteri accentati: in questi casi l'accento può essere sostituito con un apostrofo che, per il problema relativo agli apici nelle stringhe, si deve indicare con due apici. Un'istruzione come

```
writeln('Il valore è ');
```

è preferibile sostituirla quindi con

```
writeln('Il valore e'' ');
```

(che stampa *Il valore e'*).

Formattare i numeri

Quando la variabile inserita nel *writeln* è di tipo numerico è possibile specificare con quante cifre dev'essere rappresentato il numero e quante di queste devono essere decimali (nel caso il numero sia reale e non intero); per fare ciò si utilizza la sintassi

```
nome_della_variabile : numero_di_cifre [: numero_di_cifre_decimali];
```

Ad esempio:

```
writeln(n:8:6);
writeln(10+2:4)
```

Ovvero, nel primo caso il contenuto della variabile *n* verrà rappresentato con 8 cifre di cui 6 decimali; nel secondo caso verranno stampati 2 spazi vuoti e poi il numero 12.
Una istruzione come questa

```
writeln(n);
```

stamperebbe invece il valore di *n* in notazione esponenziale (ad esempio 3.4E4 anziché 34.000).

Riguardo input e output, si può anche fare un esempio con i numeri di con un programma che calcoli il quadrato di un numero reale:

```
program quadrato_di_un_numero;
var n,r:real;
uses crt;
begin
     clrscr;
     writeln('Inserisci un numero reale');
     readln(n);
     clrscr;
     r:=sqr(n);
     writeln('Il quadrato del numero inserito è',r:10:6');
     readkey;
end.
```

- il programma stampa la stringa 'Inserisci un numero reale';
- con *readln(n)* assegna alla variabile n un valore immesso da tastiera;
- alla variabile *r* viene assegnato il valore del quadrato di *n*;
- viene scritto il risultato.

Si noti che per l'assegnazione di un certo valore a una variabile si usa la sintassi:

```
nome_della_variabile := valore_assegnato_alla_variabile;
```

Questo valore può essere, come nell'esempio precedente, il risultato di un'espressione.

Librerie e funzioni predefinite

Una **libreria** non è altro che un file contenente varie procedure, funzioni e/o costanti che possono essere utilizzate una volta che il file è stato incluso all'interno del programma. Questo da un punto di vista teorico è molto utile perché permette di avere programmi molto brevi, ma da un punto di vista di sviluppo è piuttosto negativo, poiché una volta compilati anche programmi molto semplici possono essere estremamente "grandi" in dimensioni fisiche (ovvero in kByte).

Il Turbo Pascal utilizza come libreria standard *Turbo.tpl*. Questa contiene tutte le funzioni di base. Non viene mai invocata, perché lo fa in automatico il compilatore durnate la compilazione. Tuttavia, come abbiamo già visto, esistono molte altre librerie.

Dall'ultimo programma osservato si nota come deve essere fatta la dichiarazione delle librerie:

```
uses nomelibreria1, nomelibreria2, ...;
```

uses è una parola riservata, quindi non può essere usata come nome di variabile o funzione. La dichiarazione delle librerie deve precedere quella delle variabili.

Le librerie e le rispettive funzioni analizzate in questa pagina sono chiamate "**librerie predefinite**" in quanto sono di norma distribuite di default dal compliatore in uso. È infatti possibile anche creare librerie personalizzate per poter agevolare il proprio lavoro in più programmi differenti.

crt

La libreria di gran lunga più usata è sicuramente *crt*, con tutte le funzioni relative all'estetica in ambito DOS. Ma non abbiamo ancora affrontato il problema dell'utilizzo di librerie all'interno dei nostri programmi.

Probabilmente avrete notato che eseguendo i programmi implementati finora l'esecuzione terminava istantaneamente senza l'aggiunga del `readln` finale, non appena il risultato veniva mostrato sullo schermo. Questo può essere abbastanza scomodo. Ma anche per questi problemi aiuta la libreria CRT, con la funzione `readkey()` che, come abbiamo già visto, permette di leggere il primo carattere che viene immesso da tastiera.

Funzioni di crt

```
clrscr;
```

Questa funzione permette di pulire lo schermo e di posizionare il cursore in alto a sinistra in posizione (1,1)

```
cursoroff;
cursoron;
```

Queste due funzioni permettono di spegnere o accendere il cursore.

```
gotoXY(posX, posY);
```

Questa procedura permette di posizionare il cursore in una posizione precisa sullo schermo.

```
sound(hz);
nosound;
```

La funzione *sound* fa emettere alla macchina un suono alla frequenza *hz*. Per fermare questo suono bisogna assolutamente utilizzare la procedura *nosound*

```
delay(time);
```

Questa funzione fa sì che il sistema si fermi in pausa per un certo numero di millisecondi definito da *time*. L'uso della procedura `delay` è tipicamente usata quando si fa uso di suoni. La struttura d'uso generalmente è la seguente:

```
sound(440);
delay(1000);
nosound;
```

Il risultato è un LA di durata un secondo. La funzione delay() è però basata sul clock del processore per calcolare il tempo, quando è stato creato il Pascal ovviamente i processori non avevano la potenza di oggi. La funzione delay() è quindi difficile da controllare ma se si vogliono ottenere ritardi definiti con bassa precisione cronometrica si possono inserire più delay() di seguito o

sfruttare i cicli.

keypressed

Questa funzione è di tipo boolean, inizialmente è false e restituisce true appea l'utente preme un tasto. Viene spesso usata con il ciclo di tipo repeat... until:

```
repeat
   ...
   ...
until keypressed;
```

in questo modo le istruzioni vengono ripetute fino a quando l'utente non preme un tasto.

readkey

Funzione che permette di leggere un carattere inserito da tastiera senza visualizzarlo. Restituisce un valore di tipo char. Per receperare il valore letto si utilizza la notazione:

```
x:=readkey;
```

Se si utilizza dopo "keypressed", si legge direttamente il carattere premuto per fare diventare "keypressed" false. Esempio:

```
repeat
   ...
   ...
until keypressed;
x:=readkey;
```

a questo punto l'utente ha premuto un tasto, con il quale è uscito dal ciclo e ha assegnato il carattere char alla variabile X.

textcolor(*numero*);
textbackground(*numero*);

Permettono di cambiare colore al testo e allo sfondo rispettivamente. Il codice di colore può anche essere il nome in inglese del colore. Questo perché nella libreria crt sono definite anche alcune costanti fra cui proprio quelle relative ai colori. Fra l'altro si può utilizzare anche la costante blink, per creare testo intermittente. Questo uso deve essere fatto così:

```
textcolor(white+blink);
```

Tuttavia, se per compilare il programma si usa Turbo Pascal, usando la unit *crt* quando viene compilato il programma, potrebbe comparire l'errore *Runtime Error 200: divide by 0* pur senza che vi sia alcuna divisione per 0 all'interno del programma. Questo è dovuto a un problema d'incompatibilità, che non si verifica però se si usa un altro compilatore.

Tabella: Le frequenze delle note

Nota	Frequenza (Hz)	Formula (Hz)
Do	262	

Do#/Reb	277	
Re	294	
Re#/Mib	311	
Mi	330	
Fa	349	
Fa#/Solb	370	
Sol	392	
Sol#/Lab	415	
La	**440**	
La#/Sib	466	
Si	494	

La frequenza di riferimento è quella del La, ovvero 440 Hz. Per ottenere le altre ottave basta raddoppiare o dimezzare le frequenze della nota corrispondente di questa ottava centrale.

graph

graph è una libreria che fornisce la possibilità di integrare la grafica nelle applicazioni Pascal.

Funzioni di graph

Prima di iniziare a disegnare, è necessario entrare in modalità grafica tramite le funzioni:

```
scheda := detect;
initgraph(scheda, modo, percorso);
```

dove scheda e modo sono due variabili da dichiarare come integer nell'intestazione del programma. La funzione `detect` ricerca la scheda grafica in uno e restituisce il valore numerico ad essa associato; `initgraph` avvia la modalità grafica utilizzando la scheda identificata dal valore numerico di `scheda` e attribuisce alla variabile `modo` la modalità grafica selezionata automaticamente da Pascal; `percorso` è una stringa che identifica il percorso nel quale sono presenti i driver della scheda grafica: normalmente è sufficiente indicare il percorso di installazione del compilatore, ad esempio C:\TP per Turbo Pascal e C:\FPC per Free Pascal.

Quando la modalità grafica viene attivata, viene aperta una finestra aggiuntiva che sarà la schermata di output delle funzioni grafiche. Per chiuderla, usiamo la funzione

```
closegraph;
```

Per usare i colori abbiamo a disposizione le funzioni:

```
setcolor(colore) {imposta il colore di primo piano}
```

Per identificare un colore possiamo usare un numero da 0 a `getmaxcolor` (è una funzione che restituisce il numero massimo di colore disponibile) oppure le costanti red, white, black, ecc...

Un altro concetto importante è quello di puntatore attivo, cioè la posizione nel quale si trova il puntatore grafico. Quando vengono effettuati dei disegni di punti e linee, viene spostato il puntatore attivo in una posizione particolare.

Vediamo ora una serie di funzioni di disegno:

putpixel(x, y, colore)

Traccia un punto di coordinate x, y e del colore scelto e sposta in x, y il puntatore attivo.

getpixel(x,y)

Restituisce il colre del pixel in posizione x, y.

moveto(x,y)

Sposta il puntatore attivo nella posizione x,y.

GetX
GetY

Restituiscono le coordinate del puntatore attivo.

GetMaxX
GetMaxY

restituiscono i valori massimi della x e della y (ovvero le dimensioni in pixel della finestra di output).

line(x1, y1, x2, y2)

Traccia una linea dal punto x1, y1 al punto x2, y2. Sposta il puntatore attivo su x2, y2.

linerel(dx, dy)

Traccia una linea partendo dal puntatore attivo fino alla posizione x+dx e y+dy (dove x e y sono le coordinate del puntatore).

lineto(x, y)

Traccia una linea dal puntatore attivo al punto di coordinate x,y.

circle(x, y, r)

Disegna una circonferenza di centro x, y e di raggio r. Sposta il puntatore attivo su x, y.

arc(x,y, ang1, ang2, r)

Disegna un arco di circonferenza di centro x, y e di raggio r a partire dall'angolo di gradi ang1 fino ad ang2. I gradi sono contati in senso antiorario a partire dalle ore 3. Sposta il puntatore attivo su x, y.

ellipse(x,y,ang1, ang2, rx, ry)

Disegna un arco ellise di centro x, y tra gli angoli ang1 e ang2 (per una ellisse completa usare i valori 0 e 359) con il raggio orizzontale rx e raggio verticale ry. Sposta il puntatore attivo su x, y.

graph3

Un altra libreria molto usata è *graph3* che permette l'utilizzo della grafica delle versione di Turbo Pascal 3.xx, e tramite semplici comandi si riesce a disegnare facilmente punti, linee e altre forme geometriche. Questa libreria fornisce ovviamente meno funzionalità della sua pronipote graph, ma non richiede l'uso di una scheda grafica. L'utilizzo di questa libreria sostituisce alcune funzioni della libreria *crt* come *clrscr*, che viene sotituito da *clearscreen*. Dal momento in cui si entra nella modalità grafica con il comando *graphcolormode* la grandezza dei caratteri aumenta e la risoluzione diminuisce.

Funzioni di graph3

Per attivare la modalità grafica è possibile usare due funzioni:

graphmode

permette di passare in modalità grafica in bianco e nero;

graphcolormode

è invece a colori. Dopo aver selezionato la modalità grafica a colori, è possibile scegliere una delle tavolozze di colori disponibili tramite la procedura

palette(*n*)

dove n è un intero compreso tra 0 e 3. A seconda del valore di *n*, il colore rappresentato dai diversi numeri cambia:

Tavolozza	Sfondo	Numero colore		
		1	2	3
0	nero	verde	rosso	marrone
0	nero	turchese	magenta	grigio chiaro
0	nero	verde chiaro	rosso chiaro	giallo
0	nero	turchese chiaro	magenta chiaro	bianco

plot(*x*,*y*,*c*)

Questa funzione permette di disegnare un punto di coordinate *x*, *y* di colore *c* che può che rappresenta un colore diverso a seconda della palette usata (vedi la tabella).

draw(*x1*,*y1*,*x2*,*y2*,*c*)

Questa funzione permette di disegnare una linea dal punto *x1*, *y1* al punto *x2*, *y2* di colore *c*

Istruzioni di controllo

If, then, else

In qualcuno degli esempi precedenti si è vista la sintassi con *if*,*then* e *else*: vediamo in cosa consiste. Nella pratica, questo costrutto consente la scelta fra due alternative: **se** (if) una certa condizione è vera **allora** (*then*) il programma esegue una certa istruzione, **altrimenti** (*else*) ne esegue un'altra:

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

che come abbiamo già visto permette la scelta fra un caso vero e un caso falso e in base a questo esegue il contenuto del *then* o dell'*else*; tuttavia a volte il ramo in *else* non è strettamente utile o addirittura non serve.

Quando le istruzioni che seguono il *then* o l'*else* sono più di una esse devono essere delimitate da istruzioni *begin* e *end*:

```
if condizione then
      begin
         istruzione 1;
         istruzione 2;
         ...
         istruzione n
      end
   else
      begin
         istruzione 1;
         istruzione 2;
         ...
         istruzione n
       end;
```

Si noti come le istruzioni che precedono *end* o *else* non debbano essere seguite da punto e virgola

Facciamo un esempio pratico.
Si vuole calcolare la radice quadrata di un numero: ciò è possibile nell'insieme dei numeri reali solo se il valore in ingresso è positivo. Il programma deve perciò:

- acquisire il numero
- eseguire una selezione binaria: *se* il numero è positivo,*allora* ne calcola la radice quadrata e espone il risultato;*altrimenti* scrive un messaggio di errore;

```
program radice_quadrata;
var n,r:real;
  begin
    writeln('Inserisci un numero positivo');
    readln(n);
    if n>=0 then
      begin
       r:=sqrt(n);
       writeln('La radice quadrata del numero inserito è ',r:8:3);
      end
    else writeln('La radice di un numero negativo non può essere espressa con
numeri reali');
    readln;
   end.
```

Case...of, else

Questo metodo permette di fare scelte più complesse. La sintassi è:

```
case selettore of
     valore 1 : begin
                    istruzioni;
                    end;
     valore 2 : begin
                    istruzioni;
                    end;
     ...

     valore n : begin
                    istruzioni
                    end;

 else: begin
             istruzioni;
             end;
 end;
```

Con il costrutto *case*, il valore della variabile selettore viene confrontato con il valore di ogni singolo caso; quando viene trovato un valore che soddisfa le condizioni sul selettore vengono eseguite le istruzioni relative al blocco in questione; poi il controllo passa alla prima istruzione dopo il costrutto *case*. Se nessun valore soddisfa il selettore viene eseguita l'ultimo blocco di istruzioni individuato dalla parola chiave **else**. Se la parte con *else* viene omessa e il selettore non è compreso tra nessuno dei casi in elenco, si passa anche in questo caso alla prima istruzione dopo il costrutto *case*.
Quando c'è solo un istruzione che segue un certo caso si possono omettere *begin* e *end*.
Occorre fare attenzione che il costrutto *case* termina con un *end*.

Facciamo un esempio del costrutto case:

```
case Contatore of
  1, 2, 50, 210: writeln(contatore);
  30, 60..80 : begin
                 writeln(variabile);
               end;
end;
```

alcune considerazioni sull'esempio proposto:

- è stata omesso il blocco facoltativo **else**
- sono state scritte delle liste di possibili casi
- è stato usato un range di valori x..y; al terzo rigo,*60..80* indica tutti i valori compresi tra 60 e 80 (inclusi).

Strutture Iterative

In Pascal abbiamo diversi tipi di strutture iterative (in genere vengono detti "*cicli*"):

For

Questa struttura ci permette di ripetere una determinata serie di istruzioni per un numero finito di

volte. La sua sintassi è:

```
for Contatore:=valore_iniziale to valore_finale do
     begin
       istruzione 1;
       istruzione 2;
       ...
       istruzione n;
     end;
```

Nella pratica questo costrutto consente la ripetizione di un certo gruppo di istruzioni fino a quando la variabile contatore raggiunge un certo valore: **per** (for) contatore:=valore_iniziale **fino a** (to) valore_finale **esegui** (do) un blocco di istruzioni racchiuso tra begin e end.

Per Contatore s'intende una variabile (di solito vengono usate le lettere I,L,M,N,J,K) che da un valore iniziale cresce di un certo valore ogni ciclo di istruzioni fino a che non raggiunge il valore finale, che rappresenta il valore per il quale il ciclo terminerà.

L'iterazione può anche procedere in modo che la variabile contatore decresca a ogni ciclo; in questo caso la sintassi è:

```
for contatore:=valore_iniziale downto valore_finale do
    begin
      istruzione 1;
      istruzione 2;
      ...
      istruzione n;
    end;
```

Facciamo un esempio: scriviamo un programma che calcoli la somma dei primi n numeri naturali usando un ciclo for.
Il programma deve:

- leggere n;
- tramite un ciclo for, sommare tutti i numeri da 1 a n;
- esporre il risultato.

```
program sommatoria;
var i,n,s:integer;
 begin
   s:=0;
   writeln('Inserisci n');
   readln(n);
    for i:=1 to n do
     begin
      s:=s+i;
     end;
   writeln('La somma dei primi n è ',s);
   readln;
 end.
```

Analizziamo il programma riga per riga:

- dichiarazione del programma;
- dichiarazione delle variabili: i è la variabile contatore, n funge da valore finale del ciclo for, q è di volta in volta il quadrato di i mentre s è la somma dei numeri naturali fino a n.

- inizio del programma
- la somma è inizialmente uguale a 0;
- messaggio di testo;
- il programma legge n;
- il ciclo for: *per i uguale a 1 fino a che i non sia uguale a n,esegue il gruppo d'istruzioni:*
- inizio del ciclo;
- alla variabile s viene assegnata la somma del valore di s+i; prima del ciclo s era uguale a 0, quindi s:=1;
- fine del ciclo: se i è diverso da n, il ciclo viene riavviato; altrimenti si prosegue all'istruzione successiva. Automaticamente alla fine del ciclo i viene incrementato di 1.

Vediamo cosa sarebbe successo se n fosse stato, ad esempio, 50:

- i:=2, diverso da 50, quindi il ciclo continua;
- s:=s+i, cioè s:=1+2; infatti la variabile s aveva valore 1 alla fine della prima fase del ciclo.
- reinizia il ciclo; i:=3,diverso da 50,quindi il ciclo continua;
- s:=s+1;cioè s:=3+3;
- e così via, fino a quando i sia diverso da 50.

While do

Questo ciclo corrisponde ad una struttura decisionale a condizione in testa.

La sua sintassi è:

```
while condizione do
begin
     istruzione 1;
     istruzione 2;
     ...
     istruzione n;
end;
```

Ovvero: **mentre**(while) una determinata condizione è vera **esegui**(do) le istruzioni all'interno del ciclo. Fino a quando la condizione è vera, il ciclo viene eseguito. Tra **begin** ed **end** vanno inserite le istruzioni che cambiano in qualche modo il valore di verità della condizione. Questo è molto importante perché altrimenti il ciclo si ripeterebbe all'infinito e il programma non riuscirebbe mai a giungere alle righe che seguono l'istruzione **end**. Questo è un comune errore che con un po' di attenzione può essere evitato. Se vi è una sola istruzione **begin** e **end** possono essere omessi.

Ad esempio

```
...
x:=1;
while x<>5 do
  x:=x+1;
...
```

In questo esempio viene eseguito quanto scritto all'interno del ciclo finché viene verificata la condizione, cioè ad x viene sommato 1 fino a che il suo valore non sia uguale a 5,e quindi la condizione iniziale del ciclo diventi falsa.

Repeat Until

Dopo aver parlato di If...Then, Else, e dei cicli For e While Do, chiudiamo il discorso sui cicli con Repeat...Until.

La sua sintassi e':

```
...
Repeat
  istruzione 1;
  istruzione 2;
  ...
  istruzione n;
Until condizione;
...
```

In sostanza questo ciclo ripete le istruzioni comprese tra Repeat ed Until (dall'inglese,*finché*) fino al verificarsi della condizione espressa, detta anche condizione di uscita. Anche in questo caso occorre porre attenzione al fatto che la condizione di uscita diventi vera in qualche modo, altrimenti finiamo in un loop infinito perdendo il controllo del programma.

Ad esempio:

```
...
x:=1;
repeat
x:=x+1;
until x=5;
```

In questo caso x viene sommato a 1 fino a che il suo valore non sia diverso da 5; quando x=5 si esce dal ciclo.

For, While, Repeat Until

Nel caso del ciclo **For** il numero di cicli eseguiti è noto, poiché il contatore parte da un elemento iniziale ed arriva fino all'elemento finale. Nel caso di **While** e di **Repeat Until** il numero di volte che viene ripetuto il ciclo generalmente non è noto a priori, in quanto dipende dal cambiamento che subisce la variabile che controlla la condizione. È da notare che le istruzioni del ciclo **Repeat Until** verranno eseguite almeno una volta poiché la condizione viene verificata alla fine, dopo il codice del ciclo; un ciclo **While**, essendo la sua condizione testata prima dell'esecuzione del codice associato al ciclo, potrebbe, se questa risulta falsa, non fare nulla e riprendere dalle istruzioni che si trovano dopo il ciclo. Questo può risultare utile in particolari casi mentre in altri può essere causa di bug.

Array

Vettori

Un metodo molto diffuso e comodo di rappresentazione e ordinamento di dati consiste nell'uso di una tabella.
Ad esempio:

Indice	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Valori	valore 1	valore 2	valore 3	valore 4	valore 5	valore 6	valore 7	valore 8	valore 9	valore 10

Se diamo un nome alla tabella, ad esempio *Array*, possiamo indicare il valore dell'elemento di indice 8 in questo modo:

```
Array[8] {indica l'elemento di posizione 8}
```

così come

```
Array[9] {indica l'elemento di posizione 9}
```

e così via.

Una struttura come quella dell'esempio precedente è chiamata **vettore**: a ogni posizione (1, 2, ..., 10) nella tabella corrisponde un valore (valore 1, valore 2, ..., valore 10): la prima colonna della tabella in questo caso si chiama *indice* del vettore.
Il vettore consente di identificare un gruppo di dati con un nome collettivo (*Array* dell'esempio precedente) e di poter individuare ogni singolo dato attraverso un indice racchiuso tra parentesi quadre, il quale rappresenta la posizione occupata dal dato nella tabella.

Anche in Pascal, così in altri linguaggi di programmazione, si possono strutturare i dati in maniera simile tramite gli **array**.

Gli array in Pascal

Per dichiarare un vettore in Pascal è necessario indicare le diverse posizioni possibile (normalmente si usano i numeri) a e il tipo di dato che conterrà ciascun valore dell'array (nella nostra tabella, il tipo di dato dei valori della riga **Valore**):

```
var
 nome_variabile: array[1..n] of tipo_del_dato;
```

che indica che il vettore nome_variabile avrà come posizioni i numeri da 1 a n (la notazione *x..y* dove x e y sono due valori dello stesso tipo indica tutti i valori compresi tra *x* e *y*).
È possibile anche usare come indici di un array valori char o definiti dall'utente:

```
var
 vettore: array['a'..'z'] of integer
```

è una dichiarazione valida, e si può facilmente fare riferimento al vettore con la scrittura come una normale variabile con la scrittura:

```
vettore[indice]
```

È possibile dichiarare gli array anche con un altro metodo:

```
type
  nome_del_tipo=array[1 .. n] of tipo_del_dato;
 var
  nome variabile:nome_del_tipo;
```

Per leggere e scrivere gli elementi di un vettore di N elementi in Pascal è possibile usare un ciclo

for:

```
for I:=1 to N do
readln(vettore[I]);

for I:=1 to N do
writeln(vettore[I]);
```

All'aumentare del contatore I, il programma scorre i posti del vettore.

Array bidimensionali

Prendiamo come esempio questa tabella:

	1	2	3	4	5	6	7	8	9	10
1	array[1,1]	array[2,1]	array[3,1]	array[4,1]	array[5,1]	array[6,1]	array[7,1]	array[8,1]	array[9,1]	array[10,1]
2	array[1,2]	array[2,2]	array[3,2]	array[4,2]	array[5,2]	array[6,2]	array[7,2]	array[8,2]	array[9,2]	array[10,2]
3	array[1,3]	array[2,3]	array[3,3]	array[4,3]	array[5,3]	array[6,3]	array[7,3]	array[8,3]	array[9,3]	array[10,3]

Nel vettore, il posto nella tabella è individuato da un indice; nella tabella bidimensionale, invece, da una coppia di indici. Una tabella bidimensionale è anche chiamata **matrice**.

Una vettore bidimensionale si dichiara in questo modo:

```
var
 nome_variabile:array[1 .. n,1 .. m] of tipo_del_dato;
```

oppure, analogamente all'esempio precedente

```
type
 nome_del_tipo = array[1 .. n, 1 .. m] of tipo_del_dato;
var
 nome variabile: nome_del tipo;
```

Per leggere e scrivere i dati di una matrice di N righe e M colonne in Pascal si possono usare due cicli for annidati:

```
for I:=1 to N do
 for J:=1 to M do
  readln(array[I,J]);

for I:=1 to N do
 for J:=1 to M do
  writeln(array[I,J]);
```

Metodo top-down

Per **metodo top-down** si intende una suddivisione di un problema, di un algoritmo o di un procedimento in sottoproblemi più piccoli e più semplici da implementare nel linguaggio desiderato.
Una delle comodità della scomposizione dei problemi in porzioni di codice è la loro riutilizzabilità: tramite il metodo top-down, infatti, il programmatore può definire blocchi di codice a cui è possibile fare riferimento durante il corso della programmazione.

In Pascal possiamo implementare soluzioni top-down tramite le **procedure** e le **funzioni**.

Procedure

Per procedura si intende una porzione di codice riutilizzabile che può prevedere parametri in ingresso ma non prevede parametri in uscita. La sua sintassi è:

```
procedure nome_procedura(variabili_richieste_come_parametri);
dichiarazioni
begin
istruzioni;
end;
```

Le diverse procedure del programma Pascal devono essere scritte prima del blocco del blocco begin...end che delimita il programma principale. Vediamo un esempio completo di un sottoproblema implementato in Pascal come, ad esempio, dare il benvenuto ad un utente, all'interno di un programma più lungo che ometteremo:

```
program Esempio;
var [...]
procedure Benvenuto;
var nome:string[50];
begin
    writeln('Come ti chiami?');
    readln(nome);
    writeln('Benvenuto su it.wikibooks, ', nome);
end;
(* qui incomincia il programma vero *)
begin
    centinaia righe di codice...
    (* in questa riga viene chiamata (o invocata) la procedura Benvenuto *)
    (* e viene eseguita la porzione di codice in essa contenuta *)
    Benvenuto;
    centinaia righe di codice...
end.
```

In questo semplice caso la procedura Benvenuto chiede un input all'utente e stampa un messaggio di benvenuto.
La comodità di questo semplice spezzone sta nel fatto che, durante l'esecuzione del programma vero e proprio, in qualsiasi punto è possibile eseguire la procedura Benvenuto quante volte si vuole riducendo così la mole di codice da scrivere e anche facilitando, ad esempio, la correzione di eventuali errori o la revisione del codice.
Questo metodo è ovviamente comodo nel caso di programmi molto lunghi o di sottoproblemi che si ripresentino molte volte nel corso del programma

Le variabili usate nella procedura e dichiarate quindi nella sezione di dichiarazione della procedura stessa (nel nostro esempio la variabile nome) sono chiamate variabili **locali**, in quanto non è possibile richiamarle se non dalla procedura nelle quali sono dichiarate. Nel programma vero e proprio e nelle procedure le variabili locali delle altre eventuali procedure non sono quindi utilizzabili.
Sono chiamate variabili **globali** le variabili dichiarate all'intestazione del programma principale, in quanto possono essere richiamate e utilizzate sia in ambito del programma principale stesso sia in ambito locale delle procedure.
Si noti che nel caso che due variabili con lo stesso nome sono dichiarate sia in ambito globale che in ambito locale, nelle procedure il compilatore prende in considerazione la variabile locale. Il

consiglio è comunque quello di dare nomi sempre diversi alle variabili, evitando sovrapposizioni di qualsiasi genere. Un esempio potrebbe rendere il tutto più chiaro:

```
program Variabili;
var principale:integer;
procedure proc1;
var locale1:integer;
begin
    (* da qui sono accessibili solo le variabili locale1 e principale*)
end
var locale2:integer;
begin
    (* da qui sono accessibili solo le variabili locale2 e principale*)
end
(* qui incomincia il programma vero *)
begin
    (* da qui è accessibile solo la variabile principale*)
end
```

Procedure con argomenti

Può risultare utile, in molti casi, passare alla procedura dei valori che specifichino meglio l'operazione da svolgere o che ne rendano l'uso più utile alle diverse situazioni. Questi valori sono detti **argomenti** o **parametri** Nell'esempio precedente, ad esempio, sarebbe meglio poter specificare ogni volta il messaggio stampato dalla procedura Benvenuto in modo tale da renderla utilizzabile in più situazioni. Vediamo l'esempio con l'uso delle variabili:

```
program Esempio;
var [...]
procedure Benvenuto (domanda, risposta: string[100]);
var nome:string[50];
begin
    writeln(domanda);
    readln(nome);
    writeln(risposta, nome);
end;
(* qui incomincia il programma vero *)
begin
    centinaia righe di codice...
    Benvenuto('Qual è il tuo nome?', 'Benvenuto nel mio sito, ');
    centinaia righe di codice...
    Benvenuto('Come si chiama la tua fidanzata?', 'Salutami allora ');
end.
```

In questo caso la procedura Benvenuto è stata chiamata due volte nel corso del programma *passando* ogni volta i due argomenti richiesti, che non sono altro che delle espressioni che verranno poi salvate nelle variabili *domanda* e *risposta*. L'output delle due procedure sarà il seguente, nel caso l'input sia Luigi o Luisa:

```
Qual è il tuo nome? Luigi
Benvenuto nel mio sito, Luigi

Come si chiama la tua fidanzata? Luisa
Salutami allore Luisa
```

I parametri sono ovviamente variabili locali della procedura.

Passaggio di parametri per valore e per riferimento

Introdotto l'uso dei parametri, è necessario però fare una distinzione molto importante tra i due modi possibili di passare una variabile:

- quando i valori sono passati per **valore** la variabile che funge da parametro assume semplicemente il valore dell'espressione introdotta nella chiamata della procedura. Questa è la situazione presentata nel programma Esempio che abbiamo precedentemente visto.
- quando i valori sono passati per **riferimento** la variabile che funge da parametro si sostituisce momentaneamente alla variabile passata nella chiamata della procedura, che verrà quindi modificata nel corso del programma. In questo caso il parametro deve essere indicato con la sintassi **`var`** `nome_var : tipo di dato;`

La differenza sarà forse più chiara con un esempio:

```
program Esempio;
var z1, z2:real;
procedure Valore (x:real);
begin
    x := 2*x
    writeln('Il valore di X è ', x);
end;
procedure Riferimento (var x:real);
begin
    x := 2*x
    writeln('Il valore di X è ', x);
end;
(* qui incomincia il programma vero *)
begin
    z1 := 5;
    z2 := 12
    Valore(z1);
    Riferimento(z2);
    writeln('Il valore di z1 è ',z1);
    writeln('Il valore di z2 è ',z2);
end.
```

Dopo l'esecuzione del programma, avremo quindi la seguente situazione:

- il valore di z1 sarà rimasto lo stesso di quando la procedura è stata invocata, in quanto il passaggio del parametro x è avvenuto per valore. Al posto di z1 si poteva passare alla procedura anche un'espressione, come `3 + 4` o anche `z1 + 5`.
- il valore di z2 dopo la chiamata della procedura Riferimento sarà pari a 24, ossia 12 * 2, in quanto il parametro x è stato passato oper riferimento e, quindi, al momento dell'istruzione `x := x*2` non varia solo la variabile x stessa all'interno della procedura ma anche il valore della variabile z2.

L'output del programma sarà quindi:

```
Il valore di X è 10
Il valore di X è 24
Il valore di z1 è 5
Il valore di z2 è 24
```

Funzioni

Il concetto di funzioni in programmazione è strettamente legato al concetto di funzione matematica.
In Pascal possiamo pensare ad una funzione come ad una procedura che restituisce un valore: le funzioni, come le procedure, devono essere dichiarate prima del programma principale e possono disporre di variabili locali e di parametri.
La loro sintassi è tuttavia leggermente diversa:

```
function nome_della_funzione(parametri):tipo_di_dato_restituito_dalla_funzione;
dichiarazioni
begin
istruzioni;
end
```

Per riferirsi al valore della funzione si deve fare riferimento al nome della funzione stessa. Creiamo ad esempio una funzione che restituisca il valore assoluto di un numero:

```
function Assoluto (x:real):real;
begin
if x<0 then
Assoluto := -x
else
Assoluto := x;
end;
```

Analizziamo il listato riga per riga:

1. la prima riga è la dichiarazione della funzione, che richiede un parametro real, *x*, e che restituisce un valore real
2. inizio della funzione (non ci sono variabili da dichiarare in questo caso perché la funzione è molto semplice)
3. se x è minore di zero allora
4. restituisci come valore l'opposto di x
5. altrimenti (x maggiore o uguale a 0)
6. restituisci il valore di x
7. fine della funzione

In questo modo, passando alla funzione 3, Assoluto restituisce 3, mentre se si passa -12 la funzione restituisce 12. È possibile in qualsiasi punto del programma in cui la funzione è stata inserita ricorrere ad essa usando la seguente notazione:

```
Assoluto(n);
```

che funge da espressione in quanto restituisce un valore.

Strumenti

Compilatori

I compilatori eseguono la traduzione del linguaggio Pascal in linguaggio macchina, creando cosi un file eseguibile.
Affianco a quelli a pagamento offerti dalla Borland sono sempre più diffusi compilatori gratuiti e open-source, sviluppati inizialmente per ambienti Linux.

A pagamento:

- Borland Pascal
- Borland Delphi

Open Source:

- Free Pascal
- GNU Pascal

Problemi di compatibilità

L'uso di differenti compilatori può far sorgere alcuni problemi di incompatibilità tra versioni, non tanto comunque per quanto riguarda il linguaggio vero e proprio.

Ad esempio, il compilatore di Delphi (che serve per creare applicazione normalmente basate su un'interfaccia grafica) necessita della direttiva di compilazione

```
{$APPTYPE CONSOLE}
```

da porre subito dopo l'intestazione del programma, la quale specifica che il programma è da eseguire in un'interfaccia solo testo della console.

Inoltre, sempre per quanto riguarda l'uso di Borland Delphi, non è necessario eseguire alcune operazioni come la cancellazione dello schermo, in quanto ciò viene fatto in automatico all'avvio del programma da esso compilato.

A seconda della versione più o meno recente del compilatore, il tipo di dato integer supporta una gamma di valori più vasta di -32.768...32.767

IDE

IDE è l'acronimo per *Integrated Developement Enviroment*, che significa ambiente di sviluppo integrato.
L'IDE è un particolare software che facilita il programmatore nella stesura, nella compilazione e nella distribuzione del programma. Un tipico esempio di aiuto offerto da un IDE è l'evidenziazione della sintassi e l'individuazione di semplici errori di sintassi prima della compilazione.

A pagamento:

- Borland Delphi
- Borland Turbo Pascal

Open Source :

- Lazarus
- Dev-Pas
- FreePascal

Da notare che Borland Delphi e Lazarus sono IDE di Delphi, ovvero un'evoluzione del Pascal orientata ad oggetti; riescono tuttavia a compilare senza problemi il Pascal normale.

Alcuni esempi commentati

Questo è un programma che, dato un numero, ne calcola la radice quadrata (senza l'uso della funzione `sqrt`, ovviamente) usando una regola spiegata durante l'esecuzione del programma.
Il cuore dell'algoritmo è un ciclo ripeti... sino a quando che, oltre a calcolare le approssimanzioni, stampa anche una tabella passo per passo dei calcoli effettuati

```
program radq;
 {$APPTYPE CONSOLE} {in alcuni compilatori è superfluo}
 const eps=1E-10;
 var  num, app: real;
begin
 clrscr;
 write('Questo programma si basa sulla regola di Newton');
 writeln(' riguardante le radici quadrate, secondo la quale ');
 writeln;
 writeln('     se app e'' un'' approssimazione della radice quadrata di num');
 writeln('     allora (num/app + app)/2 e'' un''approsimazione migliore.');
 writeln;
 write('Il programmera continuera'' a calcolare un''approssimazione migliore');
 writeln(' sino a quando non varra''' );
 writeln;
 writeln('     |num/app^2 - 1| < 10^-10');
 writeln;
 writeln('Introdurre il valore del numero num di cui si vuole calcolare la
radice quadrata e della prima approssimazione app');
 writeln;
 write('     '); readln(num);
 write('     '); readln(app);
 writeln;
 if num < 0 then
  writeln(num:10:2, ' non ha radice quadrata reale')
 else
 begin
  {stampa una piccola tabella}
  writeln('         app                          app^2');
  writeln;
  repeat
   writeln(app:20:10, app*app:30:10); {stampa la riga}
   app:=(num/app +app)/2; {calcola la successiva approssiamazione}
  until abs(num/ sqr(app) -1) < eps; {se il valore non differisce troppo da
quello reale}
  writeln;
  writeln(app:20:10, app*app:30:10); {stampa l'ultima approssimazione contente
il valore definitivo della radice quadrata}
 end;
 readln; {in alcuni compilatori è superfluo}
end.
```

Il seguente esempio è un programma che permette invece di simulare il lancio di un oggetto con velocità e posizione iniziale scelte dall'utente; per fare ciò si ricorre all'uso della libreria graph, a cui è stata dedicata una sezione nel corso del libro.
Come per il programma precedente, bisognerà adattare alcune opzioni in base al compilatore in uso.
Il cuore del programma è qui la procedura lancio, che calcola man mano le posizioni dell'oggetto lanciato e le disegna sullo schermo grafico.

```
program lanci;
uses crt, graph;
{i tipi word e double sono estensioni dei tipi di dati rispettivamente integer e
real}
```

```
{funzionano in modo analogo ma hanno un range di dati diverso}
var t,v0,x0, y0:double; a:real;
    s,m:integer;
    c:word;
    r:char;
const g = 9.8;

{questa procedura serve a cambiare sempre i colori}
procedure move_color (pos:integer);
begin
    c:=c+pos;
    setcolor(c mod getmaxcolor);
end;


procedure lancio (v0, x0, y0 :double; a:real; delay_time:integer);
{longint è un'estensione degli integer ma con più range}
var y,x, vx, vy, t:double;
    xi, yi:longint;
begin
    a:=pi() * a / 180;
    vx := v0 * cos(a);
    vy := v0 * sin(a);
    {è necessario troncare i valori perchè la grafica funziona solo con valori
interi}
    moveto(trunc(x0), getmaxy-1-trunc(y0));

    move_color(1);

    repeat
    begin
     y:=vy * t-( g/2 * t* t)+y0;
     x:=vx * t+x0;
     {0,0 corrisponde all'angolo in alto a sinistra, noi vogliamo quello in
basso a sinistra}
     {per questo invertiamo il valore della y secondo getmaxy}
     yi:=trunc(getmaxy-y);
     xi:=trunc(x);
     move_color(1);
     lineto(xi,yi);
     move_color(-1);
     circle(xi,yi,20);
     t:=t+0.25;
     delay(delay_time);
    end;
    until (yi > getmaxy); {fino a quando la "pallina" non esce dallo schermo}

    writeln('Lancio terminato. Premere un tasto per continuare');
    readkey;
end; {lancio}

begin
    s:=detect;
    initgraph(s,m,'H:\FPC'); {ovviamente bisogna cambiare la directory}
    directvideo:= true;

    repeat
    clrscr;
    begin
     writeln('       SIMULAZIONE DI LANCI INCLINATI');
     {stampa un piccolo menu}
     writeln; writeln;
     writeln('   L. Nuovo lancio');
     writeln('   C. Cancella schermo grafico');
```

```
     writeln('   E. Esci dal programma');
     writeln;writeln;
     write('               Selezionare un''opzione -> '); readln(r);
     {le diverse opzioni...}
     case r of
      'L', 'l':
      begin
         clrscr;
         writeln('Introdurre i valori della velocita'' iniziale v0 (espressa in
m/s)');
         writeln('della misura in gradi dell''angolo di inclinazione del lancio
dal suolo');
         writeln('e della posizione (x, y) iniziale del lancio');
         write('  v0      = '); readln(v0);
         write('  ang (ø) = '); readln(a);
         write('  x0      = '); readln(x0);
         write('  y0      = '); readln(y0);
         {legge le opzioni e chiama la procedura lancio}
         lancio(v0, x0,y0, a, 30);
      end;
      'c', 'C':
      begin
         {per cancellare lo schermo chiudiamo e riapriamo la sessione grafica}
         closegraph;
         s:=detect;
         initgraph(s,m,'H:\FPC');
         directvideo:= true;
      end;
     end; {case of}
    end; {until}
    until (r = 'e') or (r  = 'E');
    closegraph;
end.
```

Licenza

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this

License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the

meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- **C.** State on the Title page the name of the publisher of the Modified Version, as the publisher.

- **D.** Preserve all the copyright notices of the Document.

- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the

Modified Version under the terms of this License, in the form shown in the Addendum below.

- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

- **H.** Include an unaltered copy of this License.

- **I.** Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- **K.** For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- **M.** Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

- **N.** Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

- **O.** Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be

replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular

numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document
>
> under the terms of the GNU Free Documentation License, Version 1.2
>
> or any later version published by the Free Software Foundation;
>
> with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
>
> Texts. A copy of the license is included in the section entitled "GNU
>
> Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the
>
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.