

Intro alla programmazione Perl

Ovvero: "There's More Than One Way To Do It"

written by gaxt

./0x00 Perl

0x00.0 Cos'è il Perl?

0x00.1 Storia del Perl

./0x01 Interpretare

0x01.0 GNU/Linux,BSD,Unix

0x01.1 Sintassi

./0x02 Tipi Dati

0x02.0 Variabili Scalari

0x02.1 Array

0x02.2 Hash

0x02.3 @ARGV

./0x03 Strutture (if, for, foreach, while/until)

0x03.0 Operatori Aritmetici

0x03.1 Operatori Logici

0x03.2 Operatori di Confronto

0x03.3 struttura if/else

0x03.4 ciclo for

0x03.5 ciclo while / until

./0x04 Il Filesystem

0x04.0 File

0x04.1 Directory

./0x05 Subroutines

0x05.0 Sub

0x05.1 my

./0x06 Moduli

./0x07 FAQ

./0x08 Outro

0x08.0 References

0x08.1 Greet to:

0x08.2 About

0x08.3 PGP Key

./0x00 Perl

0x00.0 Cos'è il Perl?

Perl è un linguaggio di programmazione ideato e originariamente sviluppato "da un certo" Larry Wall.

Originariamente era stato creato per la configurazione di un potente sistema multiplatforma, risultò molto potente e flessibile, infatti da qualche anno è maggiormente utilizzato per la programmazione lato server, per la manipolazione delle stringhe, per la creazione di interfacce grafiche e per amministrare sistemi, anche se la rete resta il suo campo prediletto (cgi). E' un linguaggio che è molto vicino al linguaggio parlato (inglese) e la sua sintassi deriva da un incrocio tra il C e gli script shell unix (es. bash). Diciamo subito che Perl è un linguaggio interpretato e non compilato.

Principali linguaggi compilati:

- C/C++
- Pascal
- JAVA

Principali linguaggi interpretati :

- Perl
- Bash scripting
- Javascript
- Html

La differenza sta che nei linguaggi compilati il file sorgente viene passato al compilatore che lo trasforma in un file oggetto poi passa al linker che aggiunge le librerie richieste dal programma e crea il file eseguibile finito.

Nei linguaggi interpretati invece il codice sorgente viene interpretato da un programma che lo esegue.

Il vantaggio principale è che il codice sorgente viene per forza distribuito, lo svantaggio è che il sistema su cui viene eseguito deve avere l'inter-Prete :)

Un altro vantaggio è la totale portabilità, uno script perl è uguale sia per GNU/Linux che per *BSD che per Win32 (a parte qualche piccoli accorgimenti).

0x00.1 Storia del Perl

Perl è l'acronimo di Practical Extraction and Report Language, ed è come abbiamo detto un linguaggio interpretato scritto da Larry Wall.

A Larry era stato commissionato un software per gestire dei file di configurazione e dei log di un sistema.

Perl è nato così, come linguaggio per elaborare testi, poi si è evoluto e adesso siamo qui a parlarne.

Cronologia:

Perl v.1 : 1987

Perl v.2 : Giugno 1988
Perl v.3 : Ottobre 1989
Perl v.4 : Marzo 1991
Perl v.5 : Ottobre/Novembre 1994

./0x01 Interpretare

0x01.0 GNU/Linux, BSD, Unix

Per interpretare i nostri script in Perl dobbiamo digitare perl seguito dal nome dello script perl in questo modo:

```
[shell /]$ perl nome_script.pl
```

Se per comodità non vogliamo scrivere tutte le volte perl nomescript.pl possiamo inserire all'inizio dello script la riga:

```
#!/usr/bin/perl -w  
....codice dello script perl...
```

Adesso lo dobbiamo rendere eseguibile con:

```
[shell /]$ chmod 775 nome_script.pl
```

Ora lo possiamo eseguire come:

```
[shell /]$ nome_script.pl
```

La flag -w, anche se si può omettere, fa sì che quando interpretiamo il codice, l'interprete Perl ci comunichi quali pericoli corriamo nell'esecuzione dello script.

0x01.1 Sintassi

Le istruzioni nel linguaggio Perl terminano con il punto e virgola:

```
print "Hello, World";
```

Come impareremo poi print serve per stampare a video del testo racchiuso fra apici o doppi apici (vedremo poi la differenza).

I commenti si scrivono ante cedendo # al testo di commento.

```
# Commento 1  
# Commento 2  
# Commento n
```

Per ogni riga di commento dobbiamo specificare il simbolo sharp (o cancelletto) #.

Perl è un linguaggio Case-Sensitive, ciò scrivere:

```
print o Print non è la stessa cosa.
```

./0x02 Tipi di Dati

0x02.0 Variabili Scalari

In Perl a differenza di linguaggi come il C non crea differenza tra la dichiarazione di numeri interi, con la virgola, stringhe di testo ecc.. Esiste un solo tipo di variabile che è detta variabile scalare. Un'altra differenza con i linguaggi "tradizionali" è che non dobbiamo dichiarare le variabili prima di utilizzarle. Qualche esempio:

```
$a="Hello, World\n";  
print "$a";
```

Questo semplice script salva in una variabile scalare di nome \$a la stringa Hello, World.

Per chi mastica un pò di C sa che \n significa newline, cioè dopo aver scritto Hello, World va a capo.

Altre sequenze di escape in Perl sono per esempio:

```
\r : Return  
\t : Tab  
\b : Backspace  
\a : Beep  
\f : Formfeed  
\e : Escape  
\ : BackSlash  
\" : Doppi Apici
```

Abbiamo visto per ora solo come stampare a video il contenuto delle variabili, se invece vogliamo leggere quello che l'utente dello script scrive sulla tastiera dobbiamo usare l'istruzione <STDIN> che sta appunto per STandarD INput.

Vediamo con un semplice esempio come si fa:

```
print "Inserisci il tuo nome\n";  
$nome=<STDIN>;  
chomp ($nome);
```

```
print "Ciao $nome come stai?";
```

Nel qui presente script vediamo che quello che noi scriviamo dopo la richiesta del programma viene salvata nella variabile scalare \$nome e poi stampa a video la frase che contiene la variabile.
 .chomp (\$nome); Serve per rimuovere dalla stringa salvata il carattere di escape che otteniamo quando premiamo invio.

Un'altro esempio, Somma di due numeri:
 Es_1.pl

```
#!/usr/bin/perl
print "Inserisci nome\n";
$nome=<STDIN>;
print "Inserisci cognome\n";
$cognome=<STDIN>;
print "Il tuo nome e': $nome, il tuo cognome: $cognome";
```

Es_2.pl

```
#!/usr/bin/perl
print "Inserisci primo numero\n";
$primo=<STDIN>;
print "Inserisci secondo numero\n";
$secondo=<STDIN>;
$somma=$primo+$secondo;
print "La somma e' $somma !\n";
```

0x02.1 Array

Il perl ci permette di sfruttare un'altro tipo di variabili, gli array. Gli array sono un'insieme di variabili dello stesso tipo, cioè scalari. Capiamo meglio cosa sono con questo disegno ASCII:

```

  ////////////////////////////////////////////////////
0 | Valore posizione 0 |
1 | Valore posizione 1 |
2 | Valore posizione 2 |
3 | Valore posizione 3 |
...| ..... |
n | Valore posizione n |
  //////////////////////////////////////
```

Un array, quindi, è un insieme di variabili che possono essere o no dello stesso tipo.

(Nota: Nel C/C++ e in altri linguaggi gli elementi degli array devono essere per forza dello stesso tipo.)

Un'array si dichiara così:

```
@nome_array=("valore0","valore1","valore2");
```

Abbiamo così creato un array chiamato @nome_array che contiene 3 elementi. Possiamo accedere al contenuto dell'array con l'espressione:

```
$nome_array[2]; #ritorna valore2
```

Nota: si accede all'array con \$ perché valore2 è un dato scalare.

```
#!/usr/bin/perl
@nomi=("marco","gino","matteo")
print "ciao $nomi[1]; #stampa ciao gino
print "ciao $nomi[0]; #stampa ciao marco
print "ciao $nomi[2]; #stampa ciao matteo
```

Come possiamo vedere Perl, come altri linguaggi, conta da zero, quindi il primo elemento dell'array sarà quello con posizione 0.

Se vogliamo accedere all'ultimo valore dell'array usiamo l'espressione \$# così:

```
print $array[$#array];
```

Se invece vogliamo aggiungere un elemento ad un array procediamo così:

```
push(@array,"valore");
```

Il valore sarà messo all'ultimo posto nell'array. Per aggiungerne due o più facciamo:

```
push(@array, ("val1","val2"));
```

Un'altra funzione degli array molto importante è pop, che permette di eliminare un elemento.

```
pop(@array); # Questo comando elimina l'ultimo elemento #dell'array
```

Se queste funzioni agiscono sull'ultimo elemento dell'array, shift e unshift fanno esattamente il contrario.

Es:

```
@array=(1,2,3,4,5,6,7,8,9,56);# inizializza l'array
shift(@array);# elimina il primo elemento quindi 1
```

```
@array=(1,2,3,4,5,6,7,8,9,56);# inizializza l'array
shift(@array, 95);
```

Questa fetta di script aggiunge all'array (@array) nella prima posizione l'elemento 95.

Altre funzioni molto importanti sono sort e reverse.

Se avete programmato in altri linguaggi vi sarà capitato di incontrare la function sort, che riordina gli elementi di un'array dal più piccolo al più grande se sono dei numeri, se invece trattiamo delle stringhe le sistemerà in ordine alfabetico.

Reverse fa il contrario, in ordine decrescente i numeri e in ordine alfabetico invertito le stringhe.

```
#Es_sort
@stringhe=('Metallica','Manson','Slipknot','Korn');
@stringhe = sort @stringhe;
print @stringhe;
#lo script mette in ordine alfabetico
```

```
#Es_reverse
@num=(10,5,6,88,7,96,65);
@num = sort @num;
print @num;
# lo script riordina in ordine decrescente
```

0x02.2 Hash

Il terzo tipo di dati del perl sono gli Hash, che sono un insieme di scalari, ma a differenza degli array che sono indicizzati per numero, gli hash accedono ai dati per nome.

Ogni elemento di un hash è quindi diviso in due parti: la chiave e il valore. La chiave identifica l'elemento, il valore è il contenuto.

```
%Songs={'Wind of change'=>'Scorpions','Starway to Heaven'=>'Led
Zeppelin','Fear of the dark'=>'Iron Maiden'};
print $songs{'Wind of change'};
```

L'output del programma sarà:
Scorpions

Per rimuovere chiavi da un hash:
delete %nome_hash{chiave};

Per cancellare tutte le chiavi:
%nome_hash=();

0x02.3 @ARGV

@ARGV è una variabile speciale del perl.
 Questa contiene i parametri che l'utente passa nella riga di comando quando richiama il nostro script.
 Come vediamo ARGV è preceduta da @, quindi assomiglia ad un array.
 Possiamo dire che "@ARGV è un array dei parametri della riga di comando".
 @ARGV quindi è formata da n elementi che prenderanno il nome di \$ARGV[0], \$ARGV[1], \$ARGV[2], ecc...
 Vediamo un esempio pratico per schiarirci le idee:

```
#!/usr/bin/perl
$nome=$ARGV[0];
print "Ciao, $nome";
```

Se interpretiamo lo script così:
 [localhost]\$ perl prova.pl OverIP

Riceveremo l'output:
 Ciao, OverIP

Il prossimo programma riceverà tre valori scalari in input dalla riga di comando e poi verrà stampata a video la loro somma.

```
#!/usr/bin/perl
$primo=$ARGV[0];
$secondo=$ARGV[1];
print $primo+$secondo;
```

Con @ARGV possiamo richiamare qualsiasi cosa, più avanti quando parleremo delle subroutines lo utilizzeremo.

./0x03 Strutture (if, switch, for, while)

0x03.0 Operatori Aritmetici

Per modificare e confrontare dei dati abbiamo bisogno di operatori, ne esistono principalmente 3 tipi:

- Operatori Aritmetici
- Operatori Logici
- Operatori di Confronto

Gli operatori aritmetici sono quelli utilizzati per la somma, la sottrazione ecc...vediamo questa tabella

Funzione	Operatore	Sintassi
----------	-----------	----------

Somma	+	$\$a+\b
Differenza	-	$\$a-\b
Prodotto	*	$\$a*\b
Rapporto	/	$\$a/\b
Resto	%	$\$a\%\b

0x03.1 Operatori Logici

Gli operatori logici servono ad esempio quando si vuole intendere : se $\$a$ è vero $\$b$ è vero ma $\$c$ è falso, si potrebbe usare la struttura if/else ma sarebbe troppo scomodo.

Perl come altri linguaggi di programmazione, mette a disposizione del programmatore degli operatori detti logici che ci risolvono questo problema

Funzione	Operatore
And	&&
Or	
Not	!

0x03.2 Operatori di Confronto

Per il confronto di dati, Perl, divide in dati numerici ed alfanumerici, per il confronto di dati numerici usiamo questa tabella:

Funzione	Operatore
Uguale a	==
Maggiore di	>
Minore di	<
Maggiore Uguale a	>=
Minore Uguale a	<=
Diverso da	!=

Invece per i dati alfanumerici (stringhe o caratteri), per usa degli altri operatori di confronto:

Funzione	Operatore
----------	-----------

Uguale a	eq	
Maggiore di	gt	
Minore di	lt	
Maggiore Uguale a	ge	
Minore Uguale a	le	
Diverso da	ne	

0x03.3 if else

La struttura di selezione if else, ci permette di poter confrontare dei dati, vediamo come è la sintassi:

```
if(condizione){
    istruzione1
}
else{
    istruzione2
}
```

se la condizione è vera, fai questo, altrimenti fai quest'altro. Possiamo anche introdurre elsif, che in pratica aggiunge un'altra possibilità.

```
if(condizione1){
    istruzione1
}
elsif(condizione2){
    istruzione2
}
else{
    condizione3
}
```

Vediamo un paio di esempi:

```
# Es_1
# Programma che comunica dato un ingresso se è maggiore o # minore di 10
print"inserisci numero: ";
$n=<STDIN>;
if($n>10){
    print"MAGGIORE DI 10\n";
}
else{
    print"MINORE DI 10\n";
}
# EOF
```

```
# Es_2
# Programma che comunica se il numero inserito è
# uguale, minore o maggiore di 3
print "inserisci numero: ";
$n=<STDIN>;
if($n>3){
    print "MAGGIORE DI 3\n";
}
elsif($n==3){
    print "UGUALE A 3\n";
}
else{
    print "MINORE DI 3\n";
}
# EOF
```

0x03.4 for

La struttura di controllo for permette di ripetere una istruzione tante volte quanto abbiamo deciso, funziona così:

```
for(start;end;cosa fare){
    istruzioni...
}
```

Chiariamoci le idee con questo esempio:

```
#!/usr/bin/perl
for($i=0;$i<100;$i++){
    print $i;
}
#EOF
```

Cosa farà questo programma? Semplicemente incrementa la variabile numerica \$i di 1 ad ogni passaggio del ciclo fino a quando \$i è uguale a 100. Se lo eseguiamo verranno stampati a schermo i numeri da 0 a 100.

(Nota: Se programmate in C/C++ sarete sicuramente avvantaggiati perché la sintassi del for è praticamente identica.)

Prima abbiamo visto cosa sono e come si utilizzano gli array, vediamo di implementare gli array in un esempio di script con al suo interno un ciclo for.

```
#!/usr/bin/perl
for($i=0;$i<10;$i++){
    print "Inserisci numero: ";
    $array[$i]=<STDIN>;
}
```

```
print @array;
# EOF
```

Come vediamo in questo esempio, lo script crea una variabile contatore \$i. Ripete le istruzioni dentro le parentesi graffe tante volte quante gli abbiamo detto, poi stampa il contenuto dell'array. Come vediamo per accedere all'array abbiamo usato \$ prima del nome, poichè i dati contenuti sono di tipo scalare. Quando invece abbiamo stampato tutto l'array abbiamo fatto precedere al nome la @ perchè ci riferivamo a tutto l'array. Secondo esempio:

```
#!/usr/bin/perl
for($i=0;$i<10;$i++){
    print"Inserisci numero: ";
    $x=<STDIN>;
    push(@array, $x);
}
print @array;
# EOF
```

Lo script sopra fa la stessa cosa del primo, la differenza sta nel fatto che il dato viene prima letto e poi messo nell'ultimo posto dell'array con la funzione push. Abbiamo così confermato il proverbio principe del Perl: "There's More Than One Way To Do It", cioè "Ci sono altri modi per fare la stessa cosa".

```
$var=$var+1; equivale a $var++;
$var=$var-1; equivale a $var--;
```

0x03.5 while/until

Spesso dobbiamo ripetere delle operazioni più volte fino a che non succede qualcosa.

Usiamo il ciclo while, che tradotto significa appunto "finché la condizione è vera", la sua sintassi è questa:

```
while(condizione){
    istruzione 1;
    istruzione 2;
    istruzione n;
}
```

Until è il contrario di while, se while eseguiva le istruzioni finché la condizione è vera, until invece esegue fino a che la condizione è falsa. La sintassi è la seguente:

```
do{
    istruzione 1;
    istruzione 2;
    istruzione n;
}while(condizione)
```

```
Es_1.
#!/usr/bin/perl -w
# Esempio con WHILE
print"Inserisci password: ";
$pass=<STDIN>;
chomp $pass;
while($pass ne pazzuord){
    print"RIPROVA: ";
    $pass=<STDIN>;
    chomp $pass;
}
```

```
Es_2.
#!/usr/bin/perl -w
# Esempio con UNTIL
do{
print"Inserisci password: ";
$pass=<STDIN>;
chomp $pass;
}while($pass eq pazzuord)
```

./0x04 Il Filesystem

0x04.0 File

I file sono un tipo di dato che viene scritto su un supporto di memoria di massa (HD, Floppy, CD/DVD, nastri ecc..).

Per utilizzare i file in perl bisogna aprire un filehandle. Cioè un tipo di variabile che ci serve per lavorare sui file.

Vediamo nella pratica come facciamo ad usare un file:

Per prima cosa dobbiamo aprirlo:

```
open(nomelogico,nomefisico);
```

nomelogico o filehandle è il nome con cui nel corso del programma ci riferiremo al file, nomefisico è invece il nome vero è proprio che ha il file sull'harddisk.

Ora che abbiamo aperto il file cosa ci facciamo? Beh potremmo scriverci dentro qualcosa.

```
open(filehandle,">>file.txt");
```

Questo file adesso è aperto per la scrittura e ciò che inseriremo verrà posto infondo al file, se invece desideriamo sovrascrivere completamente il file metteremo una sola ">".

Vediamo qualche esempio:

```
#!/usr/bin/perl
open(file,">>file.txt");
print file "Hello, World!!!!";
close(file);
```

Questo stupido esempio mostra come è possibile scrivere del testo nel file, poi lo chiude.
close(filehandle) chiude il file.

Vediamo un esempio di script che somma dei numeri e stampa la loro somma nel file.

```
#!/usr/bin/perl
print "inserisci numero: ";
$a=<STDIN>;
print "inserisci numero: ";
$b=<STDIN>;
open(file,">>files.txt");
print file $a+$b;
```

0x04.1 Directory

Il filesystem organizza i file in gruppi detti Directory, le cartelle, che possono contenere file oppure altre directory.

Per utilizzare le directory abbiamo bisogno di creare un directory handle, questo è simile ad un file handle, ma con l differenza che non gestisce file ma directory.

Per aprire un directoryhandle si usa il comando:

```
opendir(dirhandle,"/directory");
Es: opendir(usr,"/usr");
```

Per leggere una directory usiamo: readdir dirhandle;

Una volta finito di usare la dir viene chiusa con: closedir dirhandle;

Vediamo un esempio:

```
#!/usr/bin/perl -w
opendir(TEMP,"/tmp");
@FILES=readdir TEMP;
closedir(TEMP);
print @FILES;
```

Questo semplice script legge tutto quello che è presente nella dir /tmp e lo copia nell'array @FILES, poi il dirhandle viene chiuso e viene stampato il contenuto dell'array.

Creare delle nuove directory è possibile mediante l'uso della funzione mkdir, la sintassi è la seguente:

```
mkdir(newdir, permessi);

#!/usr/bin/perl -w
print "Nome della dir da creare: ";
$newdir=<STDIN>;
chomp $newdir;
mkdir($newdir,0755)||die "Impossibile creare $newdir:$!";
```

Per gli utenti win bisogna usare 0755 come permesso, per gli smanettoni unix/linux il codice del permesso cambia a seconda dell'occorrenza, se vogliamo che solo il proprietario modifichi la dir mettiamo 0755, se vogliamo che tutti possano modificare e accedere alla dir mettiamo 0777, se vogliamo che nessuno possa fare niente 0000.

Per una guida alle autorizzazioni relative ai file e alle dir, cerca su google o sugli appunti di informatica libera.

per rimuovere una dir usa la funzione: rmdir nomedir;

./0x05 Subroutines

0x05.0 Sub

Le subroutines sono dei sottoprogrammi che eseguono un lavoro specifico, se dobbiamo eseguire un gruppo di istruzioni più volte, ad esempio per l'acquisizione dei dati in una rubrica, dovremmo scrivere n volte il codice ma perl, come quasi tutti i linguaggi, permette di creare dei sottoprogrammi. la sintassi è questa:

```
sub NomeSub{
    istruzione 1;
    istruzione 2;
    ...
    istruzione n;
}
```

Per richiamarla quando ci serve nel programma la eseguiamo con :

```
NomeSub();
```

```
Es_1:
#!/usr/bin/perl
sub Hello{
    print"Hello, World!";
```

```
}  
Hello();
```

0x05.1 my

Le variabili che utilizziamo nei nostri programmi hanno una validità che dura per tutto il programma, ciò significa che se assegno un valore ad una variabile questa se lo tiene fino a che lo modifico.

my permette di riferirsi alle variabile usata nella subroutine in cui è utilizzata, quindi:

```
$a=10;  
print "$a\n";  
sub prova{  
    my $a=5;  
    print $a;  
}  
prova();
```

L'output dello script sarà:

```
10  
5
```

Poiché la variabile globale \$a=10 e la variabile locale \$a=5.

./0x06 Moduli

Perl mette a disposizione del programmatore dei moduli, delle sorta di librerie, per risolvere i problemi più duri in modo semplice e soprattutto veloce.

Ad esempio, se volessimo crittografare una stringa negli algoritmi DES, RSA, Blowfish, ecc.. Dovremmo scriverci tutto l'algoritmo di crittografia da soli, il che sarebbe troooooo lungo.

Ci viene in aiuto la comunità perl che ha sviluppato dei Moduli che contengono le funzioni che ci permettono di crittare/decrittare una stringa. Moduli per Perl ce ne sono a migliaia e si possono trovare sul sito del progetto CPAN <http://www.cpan.org>.

Sul sito si trova anche una guida all'installazione sotto Win, Unix, VMS, Mac ecc..

Una volta installati i moduli possono essere richiamati quando scriviamo uno script con la sintassi:

```
use Modulo;  
#Esempio  
use Crypt::DES;
```

D'ora in poi nel codice potremo usare le function implementate nel modulo `Crypt::DES`, (`encrypt`, `decrypt`, `ecc`.)
Per ricevere informazioni su un determinato modulo installato sulla vostra macchina digitate nella shell: `perl Nome::Modulo`.

./0x07 FAQ (Frequently Asked Question)

0x07.0 Se ho Win32 (9x/ME/2k/Xp/NT) ?

Avete bisogno di 2 phile, uno e il Windows Installer e l'altro è il file di installazione dell'inter-prete.

Li trovate entrambi su <http://activestate.com>

0x07.1 Libri sul Perl?

Di libri che parlano di questo linguaggio ce ne sono molti, io consiglio lo storico *Programming Perl*, meglio noto come *Camel Book*, scritto nientemeno che da Larry Wall lo sviluppatore del linguaggio.

Un altro buon libro è *The Perl CookBook* (detto *Ram Book*), scritto da Tom Christiansen e Nathan Torkington, ma se cercate qualcosa sul sito della o'reilly troverete qualche buon e-book.

0x07.2 Cosa sono i Perl Mongers?

Perl Mongers (PM) è una organizzazione che promuove Perl e si divide in distaccamenti di zona, tipo i LUG.

Anche in italia esistono alcune di queste realtà, al momento quelle riconosciute da www.pm.org sono 6, *Bologna.pm*, *Nordest.pm*, *Salerno.pm*, *Torino.pm*, *Roma.pm* e *Pisa.pm* .

0x07.3 Come si avvia Debugger di Perl?

Per avviare il debugger facciamo:

```
perl -d NomeScript.pl
```

Per andare avanti digiatiamo: `n` (next).

0x07.4 Come scarico i moduli in win?

Per cercare, installare i moduli con windows è stata creata un'applicazione (`ppm`) implementata all'interprete `ActivePerl`.

Per usarla digita nel prompt `ppm`.

per cercare: digita `s NomeModulo`

per installarlo: `install NomeModulo`

per rimuovere il modulo dal sistema: `remove NomeModulo`

0x07.5 Come scrivo a colori?

Con il prompt del DOS non funziona, ma con la console di linux si.
Per scrivere un output a colori devo usare il modulo Term::ANSIColor, quindi:

```
#!/usr/bin/perl -w
use Term::ANSIColor;
print color("blue"), "OverIP\n", color("reset");
print color("red"), "gaxt\n", color("reset");
print color("green"), "shen139\n", color("reset");
```

L'output sarà:

```
OverIP
gaxt
shen139
```

./0x08 Outro

Questo testo non è una guida completa al linguaggio perl, (cosa vi aspettate da una 20ina di pagine?) ma solo una panoramica introduttiva e un piccolo manuale della sintassi di base.

Quindi non rompetemi le balle con email dicendo: "Che guida di merda, non hai neanche parlato delle Gtk e dei Socket..." o roba del genere perché verranno cestinate.....:P

Ci sono molti errori di battitura, la colpa non è mia ma di OverIP che mi ha messo fretta per la consegna...:D

0x08.0 References

- <http://www.perl.com>
- <http://www.perlmonks.org>
- <http://www.perldoc.com>
- <http://www.perl.it>
- <http://www.cpan.org>

0x08.1 Greetings

```
Greetings to: ./ OverIP,shen139,Tiger87 e tutta l'hacklab crew...
              ./ Izzy e tutti quelli di #segnalati
              ./ peeeter, angusyong, Evilsoul
              ./ invisibleknow,styx^,enigma39139_{}
```

0x08.2 About

Mail: [gaxt <at> hacari <dot> org](mailto:gaxt@hacari.org) (text only)

Web: <http://colander.altervista.org>

IRC: **server:** Azzurra **chan:** #hacklab #segnalati #colander
#peeter

Music: Marilyn Manson (The Golden Age of Grotesque)
System of a Down (Toxicity)
Misfits (American Psycho)

0x08.3 PGP Key

Bits/KeyID User ID
1024/5BE4A0D9 gaxt <gaxt@hacari.org>

-----BEGIN PGP PUBLIC KEY BLOCK-----

Version: 2.6.3i

```
mQCNA0EHv4AAAAEEAMAqk04CG3SLJ2ABo0Dj28y5jgh2TAHER5rTOkx2YGNTto+nz
EC5IcAMX4MA2Ta7ue48U4ux/ew5TqPhtx65xHl4PTLb1DmZ2B/9Yuwnd30Xb7kM4
wP+9b43ioHwuc8qRvJjMOSFwxZmM0ZobuSescB3/xnWwneK7QY/vKxdb5KDZAAUR
tBZnYXh0IDxnYXh0QGhhY2FyaS5vcmc+iQCVAwUQQQe/gI/vKxdb5KDZAQGZKwP9
GsBECcDiK6NxZjh1XjHhCmiV9/MTLnhcg2VTZZKWtY8MUZ3BoOaGROxNorDnr3Qy
j0ANPqHB3GfGCYsZpAIvgrHwNjTgkVZZ+n7KSf9ayDE23mcYvhwdqW1eNe2YgKyB
tgPVeEbNQZ7rP6qzOJRU6ILtuPcAN7Uwf0z5mMnAUJI=
=oh0N
```

-----END PGP PUBLIC KEY BLOCK-----

FingerPrint: B1 40 C3 D8 7E 98 2D D7 6E D0 BE F7 C2 AC 33 33