
IL LINGUAGGIO C

-

> Guida pratica alla
programmazione

Autore: **BlackLight**
< blacklight@autistici.org >
rilasciato sotto licenza GNU GPL 3, copyleft 2005-2008

Indice

Cenni di programmazione.....	8
.....	8
Il programma	8
Breve storia della programmazione	8
I linguaggi a basso livello.....	8
I linguaggi a medio/alto livello.....	8
Il C.....	9
L'evoluzione ad oggetti del C - il C++.....	10
La programmazione oggi.....	10
Cosa serve per programmare in C.....	12
.....	12
Struttura di un programma in C e cenni su linguaggi compilati e interpretati.....	13
.....	13
Linguaggi compilati e interpretati	13
Note.....	14
Il primo programma.....	15
.....	15
Uso delle variabili.....	17
.....	17
Tipi di variabili	17
Operazioni elementari sulle variabili.....	18
Stampa dei valori delle variabili.....	19
Variabili locali e globali.....	20
Variabili static e auto.....	21
Costanti: l'istruzione #define e la keyword const.....	22
Variabili register e volatile.....	23
Funzioni e procedure.....	24
.....	24
Definizione intuitiva di funzione.....	24
Esempi d'uso di funzioni e standard di utilizzo.....	25
Procedure.....	27
Funzioni statiche.....	28
Funzioni globali/locali	29
Input da tastiera.....	30
.....	30
Controllare il flusso di un programma.....	32
Cicli if-else.....	32
Operatori di confronto.....	33
Operatori logici.....	34
Strutture switch-case.....	37
Cicli iterativi - Istruzione for.....	39
Cicli iterativi - Istruzione while.....	41

Cicli iterativi - Istruzione do-while.....	42
Istruzione goto.....	43
Istruzione break e continue.....	44
Gli array.....	45
.....	45
Array monodimensionali.....	45
Matrici e array pluridimensionali.....	47
I puntatori.....	49
.....	49
Strutture dinamiche.....	49
Liste monolanciate.....	49
Liste circolari.....	50
Alberi e Grafi.....	50
Puntatori in C.....	51
Passaggio di puntatori alle funzioni.....	52
Puntatori e array.....	54
Passaggio di array a funzioni.....	55
Allocazione dinamica della memoria.....	55
Puntatori a funzioni.....	57
Funzioni di callback.....	57
Stringhe.....	59
.....	59
Dichiarazione di una stringa.....	59
Operare sulle stringhe - La libreria string.h.....	62
strcmp.....	62
strncmp.....	63
strcpy.....	63
strncpy.....	64
strcat.....	64
strncat.....	65
strstr.....	65
Altre funzioni sulle stringhe.....	66
sprintf.....	66
snprintf.....	66
sscanf.....	67
gets.....	67
atoi.....	68
Argomenti passati al main.....	68
Uso delle stringhe e sicurezza del programma.....	69
Algoritmi di ordinamento.....	74
.....	74
Naive sort.....	74
Bubble sort.....	75
Insert sort.....	77

Quick sort.....	78
Tipi di dato derivati, enumerazioni e strutture.....	80
.....	80
Definire propri tipi - L'operatore typedef.....	80
Enumerazioni.....	81
Dati strutturati.....	82
Direttive per il preprocessore.....	85
.....	85
La direttiva #include	85
La direttiva #define	86
Controllo del flusso	86
Macro con parametri	88
Macro predefinite	89
Operatori # e ##	89
Direttive #error e #warning	90
Funzione ricorsive.....	91
.....	91
Esempio informale di ricorsione.....	91
Esempio pratico di ricorsione.....	92
Ricorsione tail e non-tail.....	93
Liste.....	95
.....	95
Liste come tipi di dato astratto.....	95
Rappresentazione statica.....	96
Rappresentazione dinamica.....	98
Gestione dei file ad alto livello.....	101
.....	101
Apertura dei file in C.....	102
Scrittura su file testuali - fprintf e fputs.....	103
Lettura di file testuali - fscanf e fgets.....	105
Scrittura di dati in formato binario - fwrite.....	108
Lettura di dati in formato binario - fread.....	109
Posizionamento all'intero di un file - fseek e ftell.....	110
Prendere parametri da riga di comando.....	112
.....	112
Libreria math.h.....	113
.....	113
Funzioni trigonometriche.....	113
Funzioni iperboliche.....	113
Funzioni esponenziali e logaritmiche.....	113
Potenze e radici.....	113
Arrotondamento e valore assoluto.....	113
Costanti.....	114
Generazione di numeri pseudocasuali.....	114

Libreria time.h.....	115
.....	115
time_t	115
struct tm	116
Esempio	116
Gestione dei file - primitive a basso livello.....	118
.....	118
File pointer e file descriptor.....	118
open.....	119
Modalità di apertura.....	119
Permessi.....	120
close.....	120
read e write.....	121
Esempio pratico.....	121
lseek.....	122
Redirezione.....	123
Gestione del filesystem a basso livello.....	124
Gestione delle directory.....	124
Socket e connessioni di rete in C.....	127
.....	127
Protocolli TCP e UDP.....	127
Indirizzi IP e endianness.....	128
Porte.....	129
Inizializzazione dell'indirizzo.....	130
Creazione del socket e connessione.....	131
Lettura e scrittura di informazioni sul socket.....	132
Lato server.....	132
Esempio pratico.....	133
Multiprogrammazione - programmazione multiprocesso e multithread.....	138
.....	138
Introduzione ai sistemi multiprogrammati.....	138
Algoritmi di scheduling.....	138
Programmazione multiprocesso.....	140
Comunicazione tra processi. Concetto di pipe.....	143
Interruzione di un processo. Concetto di segnale.....	146
Programmazione multithread.....	148
Programmazione della porta parallela in C.....	151
.....	151
Disclaimer.....	151
Struttura della porta.....	153
Individuazione dell'indirizzo della porta parallela.....	153
Primitive di sistema per la programmazione del dispositivo.....	154
ioperm.....	154
inb o outb.....	155

Esempio pratico.....	155
Interfacciamento tra C e MySQL.....	157
.....	157
Applicazione pratica.....	157
CGI in C.....	163
.....	163
Pagine statiche e pagine dinamiche.....	163
Richieste GET e POST.....	165
GET.....	165
POST.....	169
Link esterni.....	171
Catturare pacchetti con le librerie PCAP.....	172
.....	172
Compilare e linkare programmi con le librerie PCAP.....	172
Trovare un'interfaccia di rete.....	172
Sniffing.....	174
Packet injection.....	177
Introduzione ai sistemi fuzzy e alle reti neurali.....	179
.....	179
Prerequisiti matematici	179
Sistemi fuzzy	179
Introduzione alle reti neurali	179
Struttura di una rete neurale	180
Tecniche di apprendimento	184
Sviluppo di una rete neurale	185
Riferimenti bibliografici	194
Raw socket.....	196
.....	196
Monitorare modifiche ai file tramite inotify.....	200

Cenni di programmazione

.

Il programma

Si definisce "[programma](#)" qualsiasi sequenza di istruzioni scritte in linguaggio macchina (l'unico linguaggio comprensibile ad un calcolatore, le famose sequenze di 0 e 1) atta ad essere elaborata da un calcolatore o comunque da una struttura informatica.

Ogni volta che accendiamo il PC facciamo uso di programmi. Word e Outlook sono programmi. Winamp è un programma. [Windows](#) stesso non è altro che un programma (un "programma di base" o "software di base"). Anche i [virus](#) sono dei programmi eseguibili.

Si pone qui il problema di *come* scrivere un programma. Per questa esigenza si fa ricorso ai [linguaggi di programmazione](#).

Breve storia della programmazione

I linguaggi a basso livello

In principio, quando i computer erano enormi mobili chiusi a chiave dentro le stanze delle università o dei laboratori di ricerca, era il programmatore stesso a dettare le istruzioni binarie al calcolatore. Ma scrivere programmi usando sequenze di 0 e 1 non è certo la cosa più comoda del mondo, oltre ad essere del tutto innaturale per l'uomo.

Passiamo alla metà degli anni 50: i programmatori creano un sistema più "comodo" per creare i loro progetti, l'Assembly. L'[Assembly](#) (o GLI Assembly, dato che la sintassi di questo linguaggio cambia in funzione di molte variabili, quali il sistema operativo in uso, l'architettura della macchina che si va a programmare e l'assemblatore in uso) non è altro che una rappresentazione simbolica del linguaggio macchina, dove ad ogni istruzione binaria corrisponde un'istruzione mnemonica, relativamente più semplice da ricordare. Così la sequenza 1100 1101 0010 0001, o CD 21 (linguaggio macchina) diventa int 21h (sintassi Assembly [Intel](#)).

I linguaggi a medio/alto livello

Ma l'[Assembly](#) non è il massimo della vita. E' pur sempre un linguaggio orientato

alla macchina, non al problema, e, col passare degli anni i progetti si fanno via via più grandi, e l'Assembly da solo non può gestirli. Ecco quindi che compaiono sulla scena i primi linguaggi ad "alto livello", ossia linguaggi più orientati al problema (e quindi all'utente) che all'architettura intrinseca della macchina. Sono il [COBOL](#), il [BASIC](#) e il [FORTRAN](#), linguaggi molto più semplici dell'Assembly ma non altrettanto potenti. Anche questi linguaggi hanno le loro pecche: il [COBOL](#) ha delle regole sintattiche troppo rigide per essere usato dalle masse di programmatori, ed è dedicato ad uno scopo, ovvero la gestione di basi di dati piuttosto primitive e di applicazioni per il business, che non lo rende universale. Il [FORTRAN](#) è ottimo per la progettazione di applicazioni a carattere scientifico, ma non per la progettazione di codice di sistema, il [BASIC](#), nonostante la sua incredibile facilità di apprendimento, non è potente, e non ha una struttura vera e propria. Inoltre, questi tre linguaggi si basano tutti sull'istruzione GOTO ("vai a"), ripresa dall'istruzione JMP dell'Assembly, quindi i programmi scritti con questi linguaggi tendono al cosiddetto "codice spaghetti", un codice spezzettato, pieno di diramazioni e salti condizionati che rendono spesso il programma illeggibile, un vero dramma quando si tratta di fare manutenzione.

Nasce quindi il [PASCAL](#), un linguaggio ad alto livello dotato di una struttura e di istruzioni vere e proprie per il controllo del flusso del programma, ma non progettato per un vasto campo di azione, quindi poco efficiente per la scrittura di codice di sistema. Al giorno d'oggi il [PASCAL](#) è usato solo per scopi didattici, grazie alla sua semplicità di apprendimento e alla sua sintassi "pulita".

Il C

Arriviamo all'inizio degli anni 70, l'hardware diventa sempre più potente e la richiesta di software cresce giorno dopo giorno, ma non esiste ancora un linguaggio ad alto livello che soddisfi qualsiasi richiesta di software. Fino al 1972, "l'ora zero" del linguaggio C: in un laboratorio della AT&T Bell [Dennis Ritchie](#) fa girare un primo prototipo del [C](#) su un DEC PDP-11 con sistema operativo [UNIX](#). Il [C](#) fu il risultato dello sviluppo di due linguaggi di programmazione più vecchi: il [B](#) (sviluppato da [Ken Thompson](#)) e il [BCPL](#) (sviluppato da [Martin Richards](#)). Per anni il linguaggio C fu strettamente legato al sistema operativo [UNIX](#) (infatti, dopo la progettazione del [C](#), tutte le successive versioni di UNIX furono scritte in questo linguaggio, e ancora oggi i [kernel UNIX](#) sono scritti in C). Nel 1989, alla luce dei vari "stili" del [C](#) formatisi, l'ANSI (American National Standards Institute) mise a punto l'[ANSI-C](#), una versione standard del [C](#) priva di ambiguità, che è in uso tuttoggi.

La novità del [C](#), ed anche il motivo di tutto il suo successo, è che il [C](#) è un linguaggio di programmazione sviluppato dai programmatori stessi, e non da un'istituzione governativa o da un'università, per i programmatori stessi: questo rende il [C](#) il *linguaggio dei programmatori*. Unita a questa caratteristica, c'è la versatilità del [C](#): un linguaggio usato tanto per semplici programmi didattici, tanto per programmare sistemi operativi: è un linguaggio che si presta ad un'infinità di usi, grazie anche ad

una libreria davvero vastissima. Il [C](#) infatti, a differenza degli altri linguaggi di programmazione, ha davvero pochissime keyword (parole riservate), ma una vastissima gamma di funzioni che spaziano dalle funzioni per l'I/O standard alle funzioni matematiche, dalla manipolazione dei [file](#) alla gestione della [memoria](#), dagli strumenti per la creazione di interfacce grafiche ([GUI](#)) alla manipolazione delle [regex](#): queste funzioni sono ormai parte integrante del linguaggio. Inoltre, chiunque può aggiungere nuove funzioni a quelle che corredano il [C](#), e questo è un altro elemento che rende il C tanto flessibile e potente.

E' inoltre uno dei pochi linguaggi ad alto livello che mette a disposizione delle funzioni per la gestione della [memoria](#) e dei [processi](#), o anche per la gestione di porte e periferiche di I/O (operazioni notoriamente di basso livello): se voglio creare un programma che gestisca queste componenti, devo fare assolutamente ricorso ad un linguaggio di basso livello come l'[Assembly](#).

L'evoluzione ad oggetti del C - il C++

Del 1982 è invece il [C++](#) che, a differenza di quello che pensano in molti, non è un nuovo linguaggio, bensì un'evoluzione naturale del [C](#); un programma scritto in [C](#) infatti verrà compilato senza problemi anche da un compilatore [C++](#). Tuttavia, il [C++](#) mette a disposizione del programmatore la [OOP](#), la [programmazione orientata agli oggetti](#), che non è qualcosa che si "aggiunge" al programma, bensì è un modo totalmente diverso di concepire un'applicazione. Quando si ha che fare con grandi progetti diventa difficile gestire l'applicazione nella sua interezza. Qui entra in soccorso il [C++](#), che permette al programmatore di gestire le singoli componenti del programma come oggetti, ossia come componenti intrinseche del linguaggio stesso; gli oggetti sono disposti in classi, che sono la loro rappresentazione astratta, ed ogni classe può ereditare oggetti da altre classi o cambiare la loro visibilità all'interno del programma (esistono oggetti privati, protetti e pubblici). La potenza della [OOP](#) permette al programmatore di fare cose davvero interessanti, come ridefinire gli operatori, fare l'[overloading](#) di funzioni, creare tipi di dati "su misura" (con i [template](#)), gestire le eccezioni in modo potente (con i blocchi try e catch). Ovviamente, occorre procedere per passi: non si possono apprezzare appieno le novità introdotte dal [C++](#) se non si ha familiarità col [C](#).

La programmazione oggi

Nel 1991 fu concepito il [Java](#) che, pur essendo un linguaggio a sè stante, è considerato da molti come un'evoluzione del [C++](#). Infatti, la sintassi di questo linguaggio deve molto al C/C++, i costrutti alla base sono molto simili. Ciononostante, il [Java](#) si pone come obiettivo quello di essere un linguaggio di programmazione orientato al [Web](#) (ed è diventato con gli anni il linguaggio di programmazione del [Web](#), grazie ai suoi meccanismi estremamente versatili di [applet](#) e [servlet](#)) e quello di essere un linguaggio multipiattaforma (write once, run everywhere è il motto del Java), e queste caratteristiche hanno decretato il suo

successo negli ultimi anni. Lo stesso programma che scrivo in [Java](#) lo posso eseguire teoricamente senza problemi su [Windows](#), su [Linux](#), un un [Mac](#) e persino su un telefonino (i famosi "giochi [Java](#)"), a patto che esista una JVM ([Java](#) Virtual Machine) per quel sistema.

Passando all'ultimo decennio, abbiamo la nascita dei linguaggi di quarta generazione, linguaggi di script (non di programmazione) estremamente intuitivi e facili da usare: il [Perl](#), il [Python](#) (entrambi sviluppati grazie al [C](#)). [Microsoft](#) ha messo a punto un suo ambiente di sviluppo basato sul [C++](#) (Visual C++) ed il [C#](#), un linguaggio ad oggetti che deve molto sia alla sintassi del [C/C++](#), sia a [Java](#).

Cosa serve per programmare in C

.

Strumenti necessari per la programmazione in C

- Un editor di testo
- Un compilatore
- Un linker

o, in alternativa,

- Un ambiente di sviluppo integrato

Come editor di testo vanno bene anche l'EDIT del DOS o il Notepad su Windows, oppure, nel caso si desideri un editor più avanzato, si può ricorrere a EditPlus o simili. Su Linux o un sistema UNIX-like, le scelte sono molte: dagli editor storici, EMACS e VI, ad editor più user-friendly (KWrite, KEdit, Kate, Gedit...). Di compilatori è possibile trovarne molti in rete, anche freeware (il compito del compilatore è quello di tradurre il vostro programma scritto in C in linguaggio macchina, creando quindi un file eseguibile). Sui sistemi Unix lo standard è il GCC, il compilatore C della GNU che trovate pre-installato in molte installazioni standard. Su Windows potete scaricare un porting gratuito di GCC per sistemi MS come DJGPP, oppure Dev-C++ (sempre GCC-based), o BCC (della Borland) oppure Visual C++.

In alternativa, potete far ricorso ad un ambiente di programmazione integrato (ossia un programma che ha già incorporato editor e compilatore); su Windows c'è Visual C++, oppure potete scaricare (gratuitamente) LCC o Rhide, un IDE basato su GCC, o lo stesso Dev-C++. Su Linux c'è l'ottimo KDevelop (per ambienti KDE), o Anjuta (per ambienti Gnome o Gtk-oriented).

Struttura di un programma in C e cenni su linguaggi compilati e interpretati

.

Un programma scritto in C ha una sua struttura particolare. In primis, i file sorgente, ossia i file che contengono il codice C per il compilatore, hanno estensione `.c`. Ci sono poi i file header (con estensione `.h`), che sono i file che contengono i prototipi per le funzioni e le variabili globali usate nel programma. Tali funzioni vengono poi implementate nei file sorgenti (quelli con estensione `.c`). Vedremo che, per usare funzioni di qualsiasi tipo in un programma C, è necessario richiamare il file header che contiene le funzioni che si vogliono usare tramite la direttiva `#include`.

I programmini contenuti in questo tutorial sono relativamente semplici, quindi possono essere contenuti in un solo file sorgente (con estensione `.c`).

Linguaggi compilati e interpretati

Il C è un linguaggio compilato: questo vuol dire che, una volta scritto il file sorgente (o i files sorgenti), occorre che questo venga passato al compilatore C assieme al nome del file in cui si desidera piazzare l'output. Con GCC faremo

```
gcc -o file_eseguibile file1.c file2.c ... filen.c
```

[\[1\]](#)

Ecco il compilatore cosa fa: Per prima cosa esegue le direttive al preprocessore (quelle che iniziano con `#`, come `#include` `#define` `#if` `#endif` `#ifdef`... alcune le vedremo nel corso di questo tutorial). Se non ci sono errori nei sorgenti, traduce il codice C contenuto nei files sorgenti in linguaggio macchina (in quanto è questo l'unico linguaggio davvero comprensibile al compilatore. In genere questo processo genera un file oggetto, con estensione `.o` o `.obj`, dove viene piazzato il codice in LM), quindi esegue l'operazione di linking, ossia crea il file eseguibile vero e proprio.

Quasi tutti i linguaggi ad alto livello (Pascal, FORTRAN, COBOL...) sono linguaggi compilati.

Il BASIC, il Perl e il Python sono invece linguaggi interpretati: ciò vuol dire che non è possibile creare un file eseguibile vero e proprio con questi linguaggi, ma, ogni volta che voglio eseguire un tale algoritmo, devo ricorrere ad un interprete, ossia un programma che traduce istantaneamente il codice ad alto livello in linguaggio

macchina.

La via di mezzo è il Java: una volta scritto un programma in Java, ho bisogno di compilarlo (ad esempio, con il comando `javac`): da questo processo ho un file con estensione `.class`, scritto in un linguaggio simile al linguaggio macchina, ma che non è linguaggio macchina. A questo punto posso eseguire il mio programma con l'interprete Java, che esegue il codice contenuto nel file `class`. E' uno dei punti di forza del Java, che lo ha reso portabile verso ogni piattaforma.

Ovviamente, i linguaggi compilati e interpretati hanno i loro pregi e difetti. Con un linguaggio compilato posso creare un file eseguibile vero e proprio, totalmente indipendente dal linguaggio, ma la procedura di precompilazione-compilazione-linkaggio è spesso molto lenta (soprattutto quando si tratta di compilare programmi dotati di GUI, di interfaccia grafica). Inoltre, il file eseguibile che ho ottenuto dalla compilazione è ottimizzato per la macchina dove l'ho compilato, non per un'altra. In poche parole, se compilo un file C su Linux, lo stesso file eseguibile non funzionerà su Windows.

Un linguaggio interpretato, invece, permette di vedere in real-time se il programma che si sta scrivendo contiene o no errori, senza a avviare la procedura di compilazione. Inoltre, un listato scritto, ad esempio, in Perl su un sistema Linux funzionerà anche se lo porto su un sistema Windows, a patto che vi sia installato l'interprete Perl. Però questi linguaggi hanno lo svantaggio di non creare un file eseguibile, ossia di non creare un vero e proprio programma da eseguire facendo un doppio click sopra.

Note

1. [↑](#) Aggiungendo `-OX` (con X variabile da 1 a 3) si otterranno diversi livelli di ottimizzazione del codice; aggiungendo `-s` verranno eliminate alcune parti non necessarie e l'eseguibile occuperà meno spazio; aggiungendo `-Wall` verranno elencati alcuni punti del programma che "potrebbero" determinare errori logici in run-time.

Il primo programma

.

Il primo programmino in C sarà un programma abbastanza semplice, che stampa sullo schermo della console "Hello world!" ed esce. Vediamo il codice:

```
/* hello.c */

#include <stdio.h>

int main(void) {
    printf ("Hello world!\n");
    return 0;
}
```

Una volta scritto questo codice con il nostro editor preferito, salviamolo come hello.c e compiliamolo con il nostro compilatore. Se usiamo GCC:

```
gcc -o hello hello.c
```

Quando lo eseguiamo (ovviamente in modalità console) apparirà la scritta "Hello world!". Ma vediamo cosa fa nel dettaglio...

Innanzitutto, la prima riga è un commento. I commenti in C iniziano con /* e finiscono con */, ma la maggior parte dei compilatori riconoscono anche i commenti in stile C++ (che iniziano con // e finiscono con la fine della riga). Esempio:

```
codice
codice /* Questo è un commento in stile C */
codice /* Anche questo è
        un commento in stile C */
codice // Questo è un commento in stile C++
```

All'interno di un commento è possibile scrivere informazioni sul programma, o commenti su un passaggio di codice eventualmente poco chiaro.

La prima vera e propria linea di codice è #include <stdio.h>: come ho accennato nel paragrafo precedente, questa è una direttiva al preprocessore, ovvero un'istruzione che dice al calcolatore che nel programma che segue si useranno le funzioni definite nel file stdio.h (i file header dovreste trovarli nella cartella include del vostro compilatore). Il file stdio.h contiene le funzioni principali per lo STanDard Input/Output, ossia le funzioni che permettono, ad esempio, di scrivere messaggi in modalità testo, di leggere valori dalla tastiera, di manipolare files e buffer... se non includessimo questa istruzione non potremmo usare la funzione printf() più avanti. Il file stdio.h è il file che useremo maggiormente nel corso di questo tutorial.

A questo punto inizia il programma vero e proprio: viene eseguito tutto ciò che si trova all'interno della funzione `main()` (la funzione principale di ogni programma), che inizia con `{` e finisce con `}`. Nel paragrafo dedicato alle funzioni vedremo meglio come funziona il `main` (scusate per il gioco di parole). Intanto anticipo che quell'int situato prima del `main()` dice al chiamante (in questo caso il sistema operativo stesso) che la funzione `main()` ritorna un numero intero (e questo spiega la riga `return 0`).

A questo punto chiamiamo la funzione `printf()`, definita nel file `stdio.h`. Questa funzione stampa un messaggio sullo standard output (la console sul monitor). Ovviamente, il messaggio è racchiuso fra parentesi tonde e i doppi apici `""`. La sequenza `\n` è una escape sequence, che dice al compilatore di andare a capo dopo aver scritto ciò che è contenuto nella `printf()` (`\n` sta per "new-line"). Ecco le principali sequenze di escape usate nel C:

- `\n` Va a capo (new line)
- `\t` Va avanti di una tabulazione (tasto TAB)
- `\b` Va indietro di un carattere (tasto BACKSPACE)
- `\a` Fa emettere un BEEP allo speaker interno (ALARM)
- `\"` Stampa i doppi apici `""`
- `\'` Stampa un apice singolo

Piccola nota: tutte le istruzioni del C finiscono con un punto e virgola `;` (in molti linguaggi di programmazione ad alto livello è così, Java, Perl, Pascal...).

L'istruzione `return 0`, come ho già detto prima, dice al programma di ritornare il valore 0 (intero) al sistema operativo e uscire. Vedremo meglio il suo funzionamento più avanti.

Uso delle variabili

.

In tutti i linguaggi di programmazione le variabili rivestono un ruolo molto importante, se non fondamentale, nella scrittura del programma. Le variabili dell'informatica sono una sorta di "contenitori" che al loro interno possono contenere numeri interi, numeri a virgola mobile, caratteri di testo ecc.

Tipi di variabili

La dichiarazione di una variabile in C (ricordando che in ANSI-C è *indispensabile* dichiarare una variabile prima di poterla utilizzare) è qualcosa del tipo

```
tipo nome_variabile;
```

Possiamo anche assegnarle un valore iniziale, in questo modo:

```
tipo nome_variabile = valore_iniziale;
```

Il tipo di variabile caratterizza la variabile stessa. Ecco i principali tipi ammessi dal C:

Tipo	Descrizione	Dimensione (in bit)
char	Caratteri di testo ASCII	8
short int	Numeri interi piccoli (da -32768 a 32768)	16
unsigned short int	Numeri positivi interi piccoli (da 0 a 65536)	16
int	Numeri interi (da -2147483648 a 2147483648)	32
unsigned int	Numeri interi positivi (da 0 a 4294967296)	32
long int	Numeri interi grandi	32
float	Numeri a virgola mobile (precisione singola)	32
double	Numeri a virgola mobile (doppia precisione, notazione scientifica)	64

Esempio:

```

int a;           // Dichiaro una variabile intera chiamata a senza
inizzializzarla
int b = 3;      // Dichiaro una variabile intera b che vale 3
char c = 'q';   // Dichiaro una variabile char che contiene il
carattere q
float d = 3.5;  // Dichiaro una variabile float d che vale 3.5
a = 2;         // Adesso a vale 2
int e = a+b;    // e vale la somma di a e b, ossia 5

```

Operazioni elementari sulle variabili

È possibile fare con le variabili ogni tipo di operazione matematica elementare: addizione (+), sottrazione (-), moltiplicazione (*), divisione (/), resto della divisione (%). Diamo però un'occhiata a questo codice:

```

int a = 2;           // Variabile int
float b = 3.5;      // Variabile float
int c = a+b;        // ERRORE!

```

Quello che ho riportato sopra è un errore molto frequente anche fra i programmatori più esperti, se provo a compilare un codice del genere otterrò un warning (un "avvertimento del compilatore, che però non pregiudica la compilazione vera e propria) o, più spesso, un errore vero e proprio (la compilazione viene arrestata). Non posso effettuare operazioni fra due tipi di variabili diversi fra loro (int questo caso, int e float): per farlo devo ricorrere alla conversione di cast, ovvero specificare esplicitamente al compilatore in che tipo di variabile voglio salvare il risultato. Esempio corretto:

```

int a = 2;
float b = 3.5;
int c = (int) a+b;    // Converto il risultato in int. c vale 5 in
questo caso

```

Oppure:

```

int a = 2;
float b = 3.5;
float c = (float) a+b; // Converto il risultato in float. c vale
5.5 in questo caso

```

A differenza delle variabili "matematiche", in C una scrittura del genere è concessa:

```

int a = a+2;    // Aggiorno il valore di a

```

Oppure, in modo più sintetico:

```

int a += 2;

```

La scrittura `a += 2` sta per `a = a+2` (sono concesse scritture come `+=` `-=` `*=` `/=` `%=`).

La scrittura `a++` è invece un incremento della variabile `a`, ed equivale a `a=a+1` (così come la scrittura `a--` equivale a `a=a-1`).

Meglio soffermarci un attimo su quest'aspetto. In C sono concesse sia scritture come `a++` sia come `++a`, ed entrambe incrementano la variabile `a` di un'unità. Qual è la differenza tra le due scritture? Una scrittura del tipo `a++` viene chiamata *post-incremento*. Ciò vuol dire che, sulla maggior parte dei compilatori, viene prima eseguita l'istruzione in cui si trova quest'operazione, quindi, alla fine dell'istruzione, la variabile viene incrementata. Una scrittura del tipo `++a` viene invece chiamata *pre-incremento*. Quando il compilatore incontra un'operazione di pre-incremento in genere incrementa prima il valore della variabile, quindi esegue l'istruzione all'interno della quale è collocata.

Se devo semplicemente incrementare il valore di `a`, è indifferente usare l'una o l'altra scrittura. Ma si osservi questo esempio di codice...

```
int a=3;
int b=4;
int c;
```

un conto è scrivere

```
c = (a++)+b;
// c vale 3+4=7
// a viene incrementata dopo l'istruzione
// ora a vale 4
```

un altro conto è scrivere

```
c = (++a)+b;
// c vale 4+4=8
// a viene incrementata prima dell'istruzione, e vale 4
```

Stampa dei valori delle variabili

È anche possibile usare le variabili in funzioni come la `printf()`. Prendete ad esempio il seguente codice:

```
int x = 3;
printf ("x vale %d",x);
```

L'output sarà:

```
x vale 3
```

La stringa di formato `%d` dice al compilatore di stampare la variabile intera posta fuori i doppi apici `""`. In questo caso stampa il valore di `x`, che è proprio 3. Se invece si desidera stampare una variabile di tipo float:

```
float x = 3.14;
```

```
printf ("x vale %f",x);
```

dove la scrittura %f dice al compilatore di stampare una variabile di tipo float. Ecco i formati di stringa principali usati per stampare i principali tipi di variabili:

Esempio:

```
/* variabili.c */  
  
#include < stdio.h >  
  
int main() {  
    int a,b,c;    // Dichiaro 3 variabili int  
  
    a = 3;  
    b = 4;  
    c = a+b;      // c vale 7  
  
    printf ("c vale %d\n",c);  
  
    a += 3;      // Ora a vale 6  
    b++;        // Ora b vale 5  
    c = a-b;    // Ora c vale -1  
  
    printf ("Ora c vale %d\n",c);  
  
    return 0;  
}
```

Il fatto interessante è che possiamo eseguire operazioni anche sulle variabili char. Le variabili char, infatti, vengono considerate dal programma come codici ASCII, ovvero ogni variabile ha il suo codice numerico (da 0 a 255) che viene convertito in runtime in un carattere (attenzione che in linguaggi come Java queste operazioni non sono concesse, in quanto i caratteri non sono in formato ASCII ma Unicode). Ecco un esempio:

```
char c = 65;  
// Equivale a scrivere char c = 'A', infatti  
// 65 è il codice per la lettera A  
  
char c += 4;    // Ora a vale E  
printf ("a = %c\n",c);
```

Variabili locali e globali

In C le variabili vanno dichiarate o all'inizio del programma o all'inizio della funzione che le usa. Attenzione: è un errore dichiarare una variabile in altri posti o usare variabili non dichiarate. Esempio:

```
int main() {
```

```

printf ("Ciao ciao a tutti!\n");

int b = 3;    // ERRORE! Non si può dichiarare una variabile
c=4;    // ERRORE! c non è dichiarata
}

```

In C++ non si ha questo errore, in quanto la dichiarazione di una variabile è considerata un'istruzione vera e propria e può essere messa ovunque, ma in C c'è l'errore.

Le variabili dichiarate all'inizio del programma (prima del main e di ogni funzione) vengono dette globali e possono essere usate da ogni funzione del programma (lo vedremo meglio quando parleremo delle funzioni), mentre le variabili locali possono essere viste solo dalla funzione che le dichiara (in C++ è anche possibile far vedere le variabili ad un solo blocco di codice). Esempio:

```

#include <stdio.h>

int var_globale = 3;    // Variabile globale

int main() {
    int var_locale = 2;    // Variabile locale
    .....
    var_globale += var_locale;    // È possibile perchè var_globale è
una variabile globale
    .....
}

```

Nel paragrafo sulle funzioni capiremo meglio il meccanismo di visibilità delle variabili globali. In genere, per questioni di modularità del codice e visibilità, è consigliabile usare le variabili globali solo quando è strettamente indispensabile. Questo perché, proprio in virtù delle sue proprietà, una variabile globale è modificabile da ogni funzione, e questo potrebbe portare a malfunzionamenti nel programma, nel caso in cui una funzione (che possiamo vedere come un 'pezzo' del programma) si trovi a lavorare su una variabile già modificata da un'altra funzione.

Variabili static e auto

Le variabili globali in genere sono statiche, ossia vengono istanziate in memoria quando il programma viene chiamato e distrutte quando il programma viene chiuso. Le variabili locali invece in genere sono automatiche, ossia vengono istanziate quando la funzione che le dichiara viene invocata e vengono distrutte quando la funzione chiamante è terminata. È però possibile stabilire se una variabile deve essere static e automatica attraverso le keyword static e auto. Esempio:

```

.....

auto int x = 7;    // Variabile automatica

```

```
int main() {
    static float pi = 3.14;           // Variabile statica
    .....
}
```

Una variabile statica non può essere esportata in sorgenti esterni a quello in cui è dichiarata. Esempio, ho una variabile definita in un file header, e questa variabile viene importata in un file sorgente in cui è incluso l'header. Non posso usare la variabile all'infuori del file "legittimo proprietario" se è dichiarata come statica.

Costanti: l'istruzione #define e la keyword const

È possibile dichiarare anche delle costanti in C, o variabili a sola lettura, delle variabili cioè che possono venire lette ma su cui non è possibile scrivere. I modi sono due:

- Attraverso l'istruzione #define:

```
#include <stdio.h>
/* Definisco la costante PI, che vale 3.14 */
#define PI 3.14

int main() {
    float area, raggio;
    .....
    area = raggio*PI*PI;
    .....
}
```

- Attraverso la keyword const:

```
.....
const float pi = 3.14;
.....
area = raggio*pi*pi;
.....
```

L'istruzione #define è, come la #include, un'istruzione al preprocessore. In poche parole, quando il compilatore incappa in una #define, legge il valore assegnato alla costante (anche se non è propriamente una costante, in quanto non viene allocata in memoria), cerca quella costante all'interno del programma e gli sostituisce il valore specificato in real-time. Ad ogni occorrenza di PI, quindi, il preprocessore sostituisce automaticamente 3.14, senza andare a cercare il corrispondente valore della variabile in memoria centrale.

Con la const, invece, creo una vera e propria variabile a sola lettura in modo pulito e veloce, e per dichiarare una costante è di gran lunga preferito quest'ultimo metodo. Ovviamente una scrittura come questa darà un errore (o un warning):

```
const float pi = 3.14;
```

```
pi += 1;
```

in quanto non è possibile modificare una variabile di sola lettura.

Variabili register e volatile

Le variabili vengono in genere allocate nella memoria RAM (sullo stack le variabili locali statiche, sullo heap quelle dinamiche, nel segmento di memoria DATA quelle globali). Ma in C è anche possibile allocare una variabile in un registro del processore (in genere l'accumulatore, EAX) attraverso la keyword register:

```
register int var_reg = 3;
```

Ovviamente, in genere è sconsigliato allocare variabili nei registri, in quanto per i registri ci passa una marea di informazioni importanti per la sessione, per il programma ecc. e la variabile può facilmente andar persa.

Dichiarando invece una variabile come volatile, questa variabile può venir modificata da altri processi o da altre parti del programma in qualsiasi momento:

```
volatile int vol_var;
```

Stringa di formato	Uso
%c	Variabili char
%d, %i	Valori in formato decimale
%x %X	Valori in formato esadecimale
%o	Valori in formato ottale
%l, %ld	Variabili long int
%u	Variabili unsigned
%f	Variabili float
%lf	Variabili double
%p	Indirizzo esadecimale di una variabile
%s	Stringhe di testo (le vedremo più avanti...)
%n	Scrivere i byte scritti finora sullo stack dalla funzione printf() (usata principalmente per format string overflow)

Funzioni e procedure

.

Ogni linguaggio di programmazione ad alto livello mette a disposizione del programmatore gli strumenti delle funzioni e delle procedure, tanto più il C, linguaggio procedurale per eccellenza.

Abbiamo già incontrato nel corso di tutorial un esempio di funzione: il `main()`. Il `main()` altro non è che una funzione speciale che viene eseguita all'inizio del programma. Ma ovviamente è possibile definire anche altre funzioni (avevo già accennato che tutto ciò che si fa in C si fa tramite le funzioni. Anche la `printf()` che abbiamo usato nei paragrafi precedenti non è altro che una funzione definita in `stdio.h`).

Definizione intuitiva di funzione

Per capire meglio come lavorano le funzioni in C, ci aiuteremo con la definizione matematica di funzione. Sappiamo che una funzione matematica è scritta in genere nella forma $y=f(x)$, ossia ad ogni valore della variabile indipendente x (che può essere o una variabile scalare, quindi una variabile a cui corrisponde un solo valore reale, o un vettore di variabili) corrisponde uno ed un solo valore della variabile dipendente y . Prendiamo ad esempio la funzione $f(x)=x+2$: ad ogni valore della x corrisponde uno ed un solo valore della funzione $f(x)$, se x è 0, $f(x)$ è 2, se x è 1, $f(x)$ è 3, e così via.

È possibile anche che in una funzione ci sia più di una variabile indipendente: ad esempio, $f(x,y)=x+y$.

Le "variabili indipendenti" delle funzioni nelle funzioni C sono i parametri, ossia i valori che si danno in input alla funzione (anche se è possibile creare funzioni senza alcun parametro), mentre il "risultato" della funzione (la "variabile dipendente") si ottiene usando la keyword `return` che abbiamo già incontrato. Ecco la struttura di una funzione in C:

```
tipo_ritornato nome_funzione(parametro1,parametro2...parametron) {  
    codice  
    codice  
    .....  
}
```


Esempi d'uso di funzioni e standard di utilizzo

Ecco un piccolo esempio:

```
int square(int x) {
    return x*x;
}
```

Questa funzione calcola il quadrato di un numero intero x . La variabile `int x` è il parametro che passo alla funzione. Ho stabilito all'inizio, dichiarando la funzione come `int`, che il valore ritornato dalla funzione (la "variabile dipendente") deve essere di tipo `int`. Attraverso la direttiva `return` stabilisco quale valore deve ritornare la funzione (in questo caso il quadrato del numero x , ossia $x*x$). In matematica, una funzione del genere la potrei scrivere come $f(x)=x^2$.

Questa funzione la posso richiamare all'interno del `main()` o di qualsiasi altra funzione del programma. Esempio:

```
int y;          // Dichiaro una variabile int
y = square(2); // Passo alla funzione square il valore 2,
              // in modo che calcoli il quadrato di 2
printf ("Quadrato di 2: %d\n",y);
```

Ovviamente, posso dichiarare un'infinità di funzioni in questo modo. Ecco ad esempio una funzione che calcola la somma di due numeri:

```
int somma(int a, int b) {
    return a+b;
}
```

Invocazione:

```
int c;
c = somma(2,3); // c vale 5
```

La maggior parte delle funzioni matematiche sono dichiarate nel file `math.h` (ci sono ad esempio funzioni per calcolare il seno, il coseno o la tangente di un numero reale, il logaritmo, la radice quadrata, la potenza n -esima...), quindi se vi interessa fare un programma di impostazione matematica date un'occhiata a questo file per capire quale funzione usare.

Ovviamente, è anche possibile creare funzioni senza alcun parametro in input. Esempio (stupido):

```
int ritorna_zero() {
    return 0;
}
```

Vediamo ora come inserire una funzione nel nostro programma. Le funzioni in C possono andare in qualsiasi parte del codice, ma l'ANSI-C, per evitare confusione, ha imposto che all'inizio del programma ci vadano i prototipi delle funzioni usate dal programma stesso. Un prototipo non è altro che la funzione vera e propria (tipo ritornato, nome e parametri) senza però la sua implementazione, ossia senza il codice fra le parentesi graffe {}. Esempio:

```
int square(int x);
```

Ecco qui un programmino d'esempio che calcola il quadrato di un numero intero stabilito attraverso la funzione square():

```
/* square.c */  
  
#include <stdio.h>  
  
int square(int x);           // Prototipo della funzione  
  
int main() {  
    int y;                   // Variabile intera  
    y = square(3);           // Ora y vale 9  
    printf ("Quadrato di 3: %d\n",y);  
                                // Più brevemente, potremo scrivere:  
                                // printf ("Quadrato di 3:  
%d\n",square(3));           // senza neanche "scomodare" la  
variabile y  
    return 0;  
}  
  
int square(int x) { // Implementazione della funzione square()  
    return x*x;  
}
```

Nei programmi di grandi dimensioni, in genere si mette il prototipo della funzione in un file header (con estensione .h), l'implementazione in un file .c e poi il programma vero e proprio nel file main.c. Esempio:

```
/* Questo è il file square.h */  
int square(int x);  
  
/* Questo è il file square.c */  
int square(int x) {  
    return x*x;  
}  
  
/* Questo è il file main.c */  
#include <stdio.h>  
#include "square.h"
```

```
// Ovviamente includo il file square.h

int main() {
    printf ("Quadrato di 4: %d\n",square(4));
    return 0;
}
```

Quando vado a compilare questo programma devo fare una cosa del genere:

```
gcc -o square main.c square.c
```

Ma per i nostri piccoli programmini non è il caso di fare una cosa del genere! Ci va tutto in un file.

Procedure

Un discorso simile a quello delle funzioni vale anche per le procedure; le procedure non sono altro che funzioni "speciali", funzioni che non hanno un valore ritornato: eseguono un pezzo di codice ed escono. Per concludere una procedura non è necessario il return (in quanto non ritorna alcun valore): al massimo ci possiamo mettere un return;. Per dichiarare una procedura userò la keyword void:

```
void hello() {
    printf ("Hello world!\n");
    return; // Questa riga è opzionale
}
```

Quando voglio chiamare questa procedura all'interno di una qualsiasi funzione, basterà fare così:

```
hello();
```

Esempio:

```
#include <stdio.h>

void hello(); // Prototipo della procedura

int main() {
    hello(); // Stampo la scritta "Hello world!"
            // attraverso la procedura hello()
    return 0;
}

void hello() { // Implementazione della procedura
    printf ("Hello world!\n");
}
```

Anche alle procedure posso passare qualche parametro. Esempio:

```
void stampa_var(int x) {
    printf ("Valore della variabile passata: %d\n",x);
}
```

Invocazione:

```
stampa_var(3); // L'output è: "Valore della variabile passata: 3"
```

Nota tecnica: attenzione a non fare cose del genere!

```
int square(int x);
double square(double x);
```

Quando vado a chiamare la funzione:

```
square(3);
```

il compilatore non sa che funzione chiamare e va nel pallone. Proprio per evitare ambiguità del genere, la maggior parte dei compilatori danno un errore (o almeno un warning) quando nel programma compaiono scritture del genere (tuttavia, nel C++ cose del genere sono possibili, con l'overloading delle funzioni, ossia con la dichiarazione di più funzioni con lo stesso nome MA con la lista dei parametri differente. In ogni caso, una scrittura come quella di sopra darà problemi anche in C++, in quanto entrambe le funzioni hanno un solo parametro e il compilatore, nel momento dell'invocazione, non sa quale funzione chiamare).

Funzioni statiche

Le funzioni statiche hanno proprietà molto simili alle variabili statiche. Tali funzioni, al pari delle corrispettive variabili, sono

- Istanziate in memoria quando il programma viene creato, e distrutte quando il processo corrispondente è terminato
- Visibili e utilizzabili *solo* all'interno del file che le ha dichiarate

La seconda proprietà impone delle limitazioni d'uso delle funzioni statiche, in modo da rendere più modulare il programma, più protetto ed evitare che qualsiasi file del programma possa richiamare qualsiasi funzione del programma.

Esempio:

```
/* file: foo.c */
#include <stdio.h>

static void fool() {
    printf ("Sono una funzione statica\n");
}
```

```

}

void foo2() {
    printf ("Richiamo una funzione statica\n");
    foo1(); // Chiamata valida. La funzione foo1() è contenuta
           // nello stesso file della funzione foo2()
}

/* file: main.c */

#include <stdio.h>
#include "foo.c"

main() {
    foo2(); // Chiamata valida. La funzione foo2() è visibile
           // al main e non è una funzione statica
    foo1(); // ERRORE! foo1() è statica
}

```

Il meccanismo della visibilità delle funzioni e delle variabili è ancora un po' primitivo nel C, e si basa tutto sul concetto di staticità, mentre verrà decisamente approfondito in linguaggi a oggetti come C++, Java e Smalltalk.

Funzioni globali/locali

C'è, infine, un'altro metodo, per creare una funzione, sconsigliato per il fatto che rende meno modulare il codice. Questo metodo consiste nel creare una funzione locale ad un'altra funzione. Ovvero una funzione visibile e richiamabile solo all'interno della funzione in cui è stata dichiarata.

Un esempio della sua creazione è:

```

#include <stdio.h>

int main(void)
{
    void hello_local_function(void) {
        printf("Local Function is Ready!\n");
    }

    printf("Richiamo la funzione interna...\n");
    hello_local_function();
    printf("Esco.\n\n");

    return 0;
}

```

Input da tastiera

.

Finora abbiamo preso in esame programmi che eseguono delle istruzioni ed escono. Ma un programma non ha molto senso se non può interagire con l'utente che lo usa. Il C mette a disposizione molte funzioni per l'I/O (Input/Output) da tastiera (quelle che useremo sono definite perlopiù in `stdio.h`). Abbiamo già incontrato la `printf()` per l'output sul monitor, ora facciamo conoscenza con la `scanf()`, per la lettura di valori dalla tastiera. Ecco la forma della `scanf()`:

```
scanf ("tipo_da_leggere",&variabile);
```

Ed ecco un piccolo esempio:

```
int a;                                // Dichiaro una variabile int

printf ("Inserisci una variabile intera: ");
scanf ("%d",&a);                      // Dico al programma di leggere il valore
impresso
```

Ecco nel frammento di programma di sopra cosa succede: Attraverso la `scanf()` dico al programma di leggere un valore intero dalla tastiera (già abbiamo visto che la sequenza `%d` dice al programma che quella che si sta per leggere o scrivere è una variabile intera) e di salvare questo valore all'indirizzo della variabile `a` (capiremo meglio questo concetto quando parleremo dei puntatori), ossia copio questo valore nella variabile intera `a`.

Ecco un programmino facile facile che somma fra loro due numeri reali:

```
/* somma.c */

#include <stdio.h>

double somma(double a, double b);      // Prototipo della
funzione somma()

int main() {
    double a,b;                        // Dichiaro 2 variabili double

    printf ("Inserire il primo numero: ");
    scanf ("%lf",&a);                  // Leggo il primo valore double e lo salvo
all'indirizzo di a

    printf ("Inserire il secondo numero: ");
    scanf ("%lf",&b);                  // Leggo il secondo valore double e lo salvo
```

all'indirizzo di b

```
    printf ("Somma fra %lf e %lf = %lf\n",a,b,somma(a,b));           //  
Stampo la somma fra a e b  
    return 0;  
}  
  
double somma(double a, double b) {  
    return a+b;  
}
```

Ecco invece un programmino che stampa l'area e la lunghezza di una circonferenza dato il raggio:

```
/* circ.c */  
  
#include <stdio.h>  
#include <math.h>  
// Includo il file math.h per poter usare la costante M_PI (pi  
greco)  
  
double area(double raggio);  
double circ(double raggio);  
  
int main() {  
    double r;                // Raggio  
  
    printf ("Inserire il valore del raggio: ");  
    scanf ("%lf",&r);        // Leggo il valore del raggio  
  
    printf ("Area: %lf\n",area(r));  
    printf ("Circonferenza: %lf\n",circ(r));  
  
    return 0;  
}  
  
double area(double raggio) {  
    return M_PI*raggio*raggio;    // pi*r2  
}  
  
double circ(double raggio) {  
    return 2*M_PI*raggio;        // 2pi*r  
}
```

Ho incluso il file math.h perché in questo file è già definita la costante M_PI (pi greco) con 20 cifre di precisione dopo la virgola.

Controllare il flusso di un programma

I programmi visti finora eseguono tutti un blocco di istruzioni all'interno del `main()`, o comunque all'interno di una funzione, ed escono. Abbiamo visto che è anche possibile interagire con il programma, ma ci manca ancora qualcosa: ci mancano gli strumenti per gestire il flusso di un programma, che esamineremo in questo paragrafo.

Cicli if-else

I cicli if-else (in inglese "se-altrimenti") sono la struttura per il controllo del programma più semplice messa a disposizione dai linguaggi di programmazione: questa struttura definisce il codice da eseguire se una data condizione si verifica e quello da eseguire se questa condizione non si verifica. La sua sintassi è la seguente:

```
if (condizione) {
    codice
    codice
}

else {
    codice
    codice
}
```

Esempio: prendiamo un frammento di codice che stabilisce se un numero intero `n` è positivo o negativo facendo uso del costrutto if-else:

```
int n; // Dichiaro n

.....

if (n>0) {
    printf ("n è positivo\n");    // Se n è maggiore di zero, allora è
    positivo
} else {
    printf ("n è negativo\n");    // Altrimenti, è negativo
}
```

Se un'istruzione if o else (o qualsiasi altro costrutto che vedremo in questo paragrafo) contiene una sola istruzione (come nel caso di sopra) si possono omettere le parentesi graffe `{}`

```
int n;
```



```

.....
if (n>0)
    printf ("n è positivo\n");
else
    printf ("n è negativo\n");

```

Dopo un'istruzione if non sempre è necessaria un'istruzione else: ecco un modo abbastanza interessante per scrivere il frammento di codice riportato sopra:

```

int n;

if (n>0) {
    printf ("n è positivo\n");
    return 0;                // Esco dalla funzione
}

printf ("n è negativo\n");   // Questa istruzione verrà
                             // eseguita se e soltanto se
                             // n è negativo, perchè se è
                             // positivo ricade nel costrutto
                             // if di sopra, che esce
                             // dalla funzione

```

Se qualcuno di voi ha programmato in Pascal, in BASIC, in Bash o in linguaggi simili avrà notato che il costrutto if del C (e dei linguaggi da esso derivati, C++, Java, Perl) manca della keyword then ("allora") usata in questi linguaggi, in quanto ridondante e inutile (bastano le parentesi graffe per stabilire dove il costrutto inizia e dove finisce).

Operatori di confronto

Abbiamo incontrato, negli esempi sopra, il simbolo di maggiore > , usato per stabilire se un valore è maggiore di un altro. Ovviamente, abbiamo anche il simbolo di minore < usato per il caso contrario. Ecco i principali operatori di confronto usati nel C:

Operatore	Significato
>	Maggiore
<	Minore
>=	Maggiore o uguale
<=	Minore o uguale
!=	Diverso
==	Uguale (Attenzione: è diverso da =)

Il simbolo == sta per "uguale" come confronto. Se ad esempio vogliamo sapere se una variabile vale 3, scriveremo:

```
if (a==3)          // NON a=3!!!
```

è invece un errore comune scrivere, nei confronti,

```
if (a=3)
```

attenzione: la scrittura di sopra fa semplicemente l'assegnamento di un valore alla variabile a. Sappiamo che il ciclo if è verificato se la condizione al suo interno è vera, viene ignorato quando la condizione è falsa. Il C prende come convenzione *vero* qualsiasi valore diverso da zero, *falso* qualsiasi valore uguale a zero. Il codice di sopra non fa altro che assegnare un valore alla variabile a ed entrare nel ciclo se il valore di a è diverso da zero (come in quest'esempio), ignorarlo in caso contrario. Il che è leggermente diverso dal fare un confronto, come volevamo noi...

In definitiva, l'uguale singolo = viene usato per gli assegnamenti (ad esempio "a=2") mentre quello doppio == per i confronti (nel Pascal invece si usa = per i confronti e := per le assegnazioni).

Operatori logici

Vediamo ora i principali operatori logici usati dal C. Facciamo prima un ripasso di logica: date due o più proposizioni logiche è possibile fare 4 operazioni fondamentali fra loro: la congiunzione (AND), la disgiunzione (OR), la disgiunzione esclusiva (XOR) e la negazione (NOT). Quando parliamo di proposizioni logiche parliamo di una qualsiasi affermazione che può essere vera o falsa. La congiunzione (AND) di due proposizioni è vera se e soltanto se entrambe le proposizioni sono vere. Ad esempio, in logica posso dire "fuori piove E Marco è uscito" solo se fuori piove E Marco è uscito, ossia solo se entrambi gli eventi sono veri. Con la disgiunzione (OR) basta invece che solo uno dei due eventi sia vero per rendere l'operazione vera. La disgiunzione esclusiva (XOR) invece richiede che un evento sia vero e l'altro sia falso per essere vera. La negazione (NOT) è, lo dice il nome stesso, la negazione di una proposizione. Se la proposizione è vera, la proposizione negata è falsa. Se "fuori piove" è una proposizione vera, "fuori non piove" è una proposizione falsa. Per maggiori delucidazione, ecco le tabelle di verità (le tabelle delle 4 operazioni logiche fondamentali), dove 0 sta per falso e 1 per vero (così come la vede la macchina. a e b sono le due proposizioni logiche su cui voglio operare):

a	b	a AND b
1	1	1
1	0	0
0	1	0
0	0	0

a	b	a OR b
1	1	1
1	0	1
0	1	1
0	0	0

a	b	a XOR b
1	1	0
1	0	1
0	1	1
0	0	0

a	NOT a
1	0
0	1

A cosa ci possono servire questi rudimenti di logica per la programmazione in C? È presto detto. Sappiamo che un computer ragiona con una logica binaria; nel processore tutte le istruzioni che noi mettiamo in un programma diventano, a livello logico-elettronico, delle semplici operazioni logiche, AND, OR, XOR e NOT. In particolare, in C useremo perlopiù tali operatori per descrivere meglio le condizioni all'interno di certi confronti, oppure per manipolare variabili a livello di bit nel caso di applicazioni di basso livello. Ecco come si scrivono in C le operazioni logiche:

Operazione	Scrittura in C
AND	&&
OR	
XOR	^
NOT	!

Vediamo qualche applicazione pratica: un frammento di codice che stabilisce se un numero è compreso fra 0 e 10. Senza operatori logici lo scriveremo così:

```
if (n>0) {
    if (n<10)
        printf ("n è compreso fra 0 e 10\n");
    else
        printf ("n è maggiore di 10\n");
} else
    printf ("n è minore di 0\n");
```

Con l'operatore logico AND scriveremo così:

```
if ((n>0) && (n<10)) {
    printf ("n è compreso fra 0 e 10\n");
}
```

Ossia: se n è maggiore di 0 E contemporaneamente n è minore di 10, allora n è compreso fra 0 e 10. Facciamo ora un esempio con l'OR: un programma che stabilisce se un numero è minore di 0 OPPURE maggiore di 10 (il contrario dell'intervallo che abbiamo visto sopra):

```
if ((n<0) || (n>10))
    printf ("n è minore di 0, oppure n è maggiore di 10\n");
```

Ossia: controlla se n è minore di 0 OPPURE è maggiore di 10. Ragionamento simile anche per lo XOR. Lo XOR è un'operazione logica molto usata in Assembly, in quanto fare lo XOR di un registro con se stesso equivale a svuotare il registro. Il NOT viene invece usato per sostituire scritte ridondanti come `n==0` o `n!=0`: infatti una variabile negata è sempre 0:

```
if (n) // Equivale a scrivere if (n!=0)
    printf ("n è diverso da 0\n");
if (!n) // Se "NOT n". Equivale a scrivere if (n==0)
    printf ("n è uguale a 0\n");
```

Gli operatori logici possono anche essere usati fra variabili, consentendo quindi di effettuare operazioni logiche fra numeri a livello di bit:

```
int a=0xa0a1a2a3;
int b = a && 0x0000ff00; // Fa un AND che azzerà tutti i byte
tranne il penultimo -> b = 0x0000a200
```

// 0 anche, esempio più immediato:

```
char a=3; // a = 00000011
char b=5; // b = 00000101
char c = a && b; // c = 00000001 = 1
```

```
// 0 ancora:

char a=3;          // a = 00000011
char b=5;          // b = 00000101
char c = a || b;   // c = 00000111 = 7
```

C'è poi l'operatore di complemento logico. Se infatti per il complemento logico usassimo **!**, noteremmo semplicemente che **(!a) == 0** se $a \neq 0$, e **(!a) != 0** se $a = 0$. Se vogliamo invece calcolare il complemento logico a 1 di un numero ricorriamo all'operatore binario **~**.

```
char a=7;   // a = 00000111
char b=~a;  // b = 11111000
```

Un'altra operazione logica messa a disposizione dal C è lo SHIFT.

Immaginiamo di avere una variabile `int i = 4`; scritta in binario (facciamo per comodità a 4 bit) sappiamo che equivale a 0100. Fare uno shift a sinistra di 1 bit (la scrittura in questo caso è `<<`) equivale a spostare tutti i bit di un posto a sinistra: la nostra variabile binaria da 0100 diventa quindi 1000, quindi `i` da 4 diventa per magia 8! Una cosa degenerate in C si scrive così:

```
int i = 4;
i = i << 1;    // Faccio lo shift a sinistra di 1 bit
```

C'è anche lo shift a destra, il simbolo è `>>`. Ad esempio, se facciamo uno shift a destra di 1 bit di `i`, questa variabile da 0100 diventa 0010, quindi da 4 diventa 2:

```
int i = 4;
i = i >> 1;    // Faccio lo shift a destra di 1 bit
```

Strutture switch-case

Le strutture switch-case sono un modo più elegante per gestire un numero piuttosto alto di costrutti if-else. Prendiamo un programmino che riceve in input un carattere e stabilisce se il carattere è 'a','b','c','d','e' oppure è diverso da questi cinque. Con l'if-else scriveremmo una roba del genere:

```
char ch;          // Carattere

printf ("Inserisci un carattere: ");
scanf ("%c",&ch);

if (ch=='a')
    printf ("Hai digitato a\n");
else {
    if (ch=='b')
        printf ("Hai digitato b\n");
    else {
        if (ch=='c')
```

```

    printf ("Hai digitato c\n");
else {
    if (ch=='d')
        printf ("Hai digitato d\n");
    else {
        if (ch=='e')
            printf ("Hai digitato e\n");
        else
            printf ("Non hai digitato un carattere compreso fra a ed
e\n");
    }
}
}
}
}
}

```

Tale scrittura non è certo il massimo della leggibilità. Vediamo ora lo stesso frammento di programma con una struttura switch-case:

```

char ch;

printf ("Inserisci un carattere: ");
scanf ("%c",&ch);

switch(ch) {
    // Ciclo switch per la variabile ch
    case 'a': // Nel caso ch=='a'...
        printf ("Hai digitato a\n");
        break; // Interrompe questo case

    case 'b':
        printf ("Hai digitato b\n");
        break;

    case 'c':
        printf ("Hai digitato c\n");
        break;

    case 'd':
        printf ("Hai digitato d\n");
        break;

    case 'e':
        printf ("Hai digitato e\n");
        break;

    default: // Nel caso il valore di ch non sia uno di
quelli sopra elencati...
        printf ("Non hai digitato un carattere compreso fra a ed e\n");
        break;
} // Fine della struttura switch-case

```

Metodo molto più pulito ed elegante. La struttura di uno switch-case è la seguente:

```

switch(variabile) {

```

```

case val_1:
    codice
    break;

case val_2:
    codice
    break;

.....

case val_n:
    codice
    break;

default: // La clausola di default non è obbligatoria
    codice
    break;
}

```

Ogni etichetta case va interrotta con la clausola break, che interrompe lo switch-case e ripassa il controllo al programma.

Cicli iterativi - Istruzione for

Immaginiamo di voler far ripetere al nostro programma un blocco di istruzioni per un tot numero di volte. Immaginiamo ad esempio un programmino che stampi dieci volte "Hello world!". Con le conoscenze che abbiamo finora, scriveremmo un lavoro del genere:

```

int main() {
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
    printf ("Hello world!\n");
}

```

Il che è decisamente scomodo. Per eventualità di questo tipo ci viene in aiuto il ciclo for, che ha la seguente sintassi:

```

for (variabile_1=valore1, ..., variabile_n=valore_n; condizione;
step) {
    codice
}

```

Dove `variabile_1,...,variabile_n` sono le cosiddette *variabile contatori*, *condizione* è una condizione booleana che stabilisce il numero di cicli da eseguire (ovvero, finché la condizione è vera esegui il ciclo `for`) e *step* l'eventuale incremento o decremento da far subire alle variabili contatore ad ogni ciclo.

Esempio chiarificatore: ecco il programmino di sopra scritto con un ciclo `for`:

```
int main() {
    int i;                                // Variabile "contatore"

    for (i=0; i<10; i++)
        printf ("Hello world!\n");

    return 0;
}
```

Dove la variabile contatore è *i*, e viene inizialmente posta, all'interno del ciclo `for`, uguale a 0. La condizione è *i<10*, ovvero *finché la variabile i è minore di 10 esegui il ciclo*, lo *step* invece è *i++*, ovvero 'ad ogni ciclo incrementa la variabile *i* (*finché, ovviamente, non varrà 10 e il ciclo può ritenersi concluso*).

Ecco un altro esempio chiarificatore:

```
int main() {
    int i;

    for (i=0; i<10; i++)
        printf ("Valore di i: %d\n",i);

    return 0;
}
```

Ecco l'output di questo programmino:

```
Valore di i: 0
Valore di i: 1
Valore di i: 2
Valore di i: 3
Valore di i: 4
Valore di i: 5
Valore di i: 6
Valore di i: 7
Valore di i: 8
Valore di i: 9
```

Ovviamente, il ciclo `for` di sopra si può scrivere in moltissimi modi:

```
for (i=1; i<=10; i++)
    printf ("Valore di i: %d\n",i);
```

In questo caso, *i* ha come valore iniziale 1 e il ciclo termina quando *i* è esattamente

uguale a 10. In questo caso l'output sarà:

```
Valore di i: 1
Valore di i: 2
Valore di i: 3
Valore di i: 4
Valore di i: 5
Valore di i: 6
Valore di i: 7
Valore di i: 8
Valore di i: 9
Valore di i: 10
```

Altro esempio:

```
for (i=10; i>0; i--)
    printf ("Valore di i: %d\n",i);
```

In questo caso, i ha come valore iniziale 10, viene decrementata di un'unità ad ogni loop e il ciclo termina quando i vale 0. L'output è il seguente:

```
Valore di i: 10
Valore di i: 9
Valore di i: 8
Valore di i: 7
Valore di i: 6
Valore di i: 5
Valore di i: 4
Valore di i: 3
Valore di i: 2
Valore di i: 1
```

Vedremo più avanti che i cicli for sono molto utili per manipolare gli array. Piccola nota: è possibile usare i cicli for anche per eseguire un blocco di istruzioni all'infinito:

```
for (;;)
    printf ("Stampa questo all'infinito\n");
```

In questo caso, dato che non c'è nessuna variabile contatore che limita il ciclo, le istruzioni all'interno del for verranno semplicemente eseguite teoricamente all'infinito. Questo perché, nonostante l'istruzione for preveda 3 campi (variabili contatore con valori iniziali, condizione di break e step), nessuno di questi 3 campi è strettamente obbligatorio.

Cicli iterativi - Istruzione while

I cicli while, o di iterazione per vero, sono cicli che eseguono un blocco di istruzioni finché una condizione specificata risulta vera. La loro sintassi è la seguente:

```
while (espressione_booleana) {
```

```
    codice
}
```

Esempio molto semplice:

```
int i=0;

while (i<10) {
    printf ("Valore di i: %d\n",i);
    i++;
}
```

Sotto un punto di vista pratico, questo frammento di codice è esattamente uguale a quello esaminato sopra, nel paragrafo sul for. Semplicemente, controlla se la variabile *i* è minore di 10: se lo è, allora esegue il blocco di istruzioni all'interno del while (ovviamente, ad ogni loop la variabile *i* viene incrementata di un'unità). Quando la condizione di partenza non è più vera, allora il ciclo termina. Esempio un po' più complesso:

```
int n;

while (n!=0) {
    printf ("Inserisci un numero (0 per finire): ");
    scanf ("%d",&n);

    printf ("Numero inserito: %d\n",n);
}
```

In questo caso, il programma mi chiederà di inserire un numero intero e stamperà il numero che ho appena inserito: se il numero è proprio 0, allora il ciclo termina (l'espressione while (*n*!=0) sta per "mentre *n* è diverso da 0").

Cicli iterativi - Istruzione do-while

Una caratteristica dei cicli while è quella che prima verificano la condizione, poi eseguono il codice contenuto al loro interno. Se la condizione iniziale è falsa a priori, il codice non verrà mai eseguito. Esempio:

```
int n = -1;           // Variabile int

while (n>0)
    printf ("Questo codice non verrà mai eseguito\n");
```

L'istruzione printf() contenuta all'interno del while non verrà mai eseguita, in quanto la condizione di partenza è falsa (il valore di *n* è minore di 0). Se volessimo che il nostro programma esegua prima il codice e poi controlli la verità della condizione dobbiamo usare un ciclo do-while. La sua struttura è la seguente:

```
do {
    codice
```

```
    codice
    .....
} while(condizione_booleana);
```

Esempio:

```
int n = -1;           // Variabile int

do {
    printf ("Questo codice verrà eseguito una sola volta\n");
} while(n>0);
```

In questo caso il programma esegue prima l'istruzione `printf()`, quindi controlla la condizione specificata. Dato che in questo caso la condizione è falsa, il ciclo termina.

Istruzione goto

L'istruzione `goto` ("vai a") è l'istruzione per i cicli più elementare, e deriva direttamente dall'istruzione `JMP` (JuMP) dell'Assembly. La sua sintassi è la seguente:

```
etichetta:
codice
codice
.....
goto etichetta;           // Salta all'etichetta specificata
```

Esempio: prendiamo il classico programmino che stampa 10 volte "Hello world!". Con l'istruzione `goto` verrebbe più o meno così:

```
int main() {
    int i=0;           // Variabile contatore

hello:               // Etichetta "hello". Ma posso chiamarla in
qualsiasi altro modo
    printf ("Hello world!\n");
    i++;             // Incremento la variabile contatore

    if (i<10)
        goto hello; // Se i è minore di 10 salto all'etichetta "hello"

    return 0;
}
```

Tuttavia, l'istruzione `goto` oggi giorno è estremamente sconsigliata, in quanto tende a creare il cosiddetto "codice a spaghetti", ossia un codice spezzettato, pieno di salti e quindi difficile da leggere. In genere i cicli `for`, `while` e `do-while` sono molto più leggibili di codici scritti con il `goto`.

Istruzione break e continue

È possibile manipolare i cicli attraverso le istruzioni break (che abbiamo già incontrato quando abbiamo parlato delle strutture switch-case) e continue. Un'istruzione break termina un ciclo, un'istruzione continue fa eseguire il codice che viene dopo. Esempio:

```
int i=0;                // Variabile "contatore"

for (;;) {              // Questo ciclo durerebbe teoricamente
all'infinito
    printf ("Ora i vale %d\n",i);
    i++;

    if (i>5)
        break;          // Se i è maggiore di 5 interrompo il ciclo
    else
        continue;      // Altrimenti lo continuo
}

printf ("Ora il ciclo è concluso!\n");
```

L'output sarà il seguente:

```
Ora i vale 0
Ora i vale 1
Ora i vale 2
Ora i vale 3
Ora i vale 4
Ora i vale 5
Ora i vale 6
Ora il ciclo è concluso!
```

In poche parole, la clausola break interrompe il ciclo che altrimenti sarebbe infinito. È possibile usare queste clausole (tra l'altro abbiamo già incontrato il break nello switch-case) in qualsiasi punto di un ciclo per interromperlo o continuarlo, al verificarsi di determinate condizioni.

Gli array

•
Gli array, o vettori, sono le strutture di dati più elementari in informatica, del tutto simili ai vettori trattati dall'algebra lineare.

Array monodimensionali

Si tratta di un insieme di variabili dello stesso tipo e accomunate dallo stesso nome (il nome dell'array). Ciò che distingue un elemento dell'array da un altro è l'indice, ovvero il suo numero, la sua posizione all'interno dell'array. Possiamo immaginare un array come una cassetiera: per sapere dove mettere le mani per trovare qualcosa ci serve il numero del cassetto dove cercare (prima cassetto, secondo cassetto...). Così, un array è una raccolta di variabili dello stesso tipo sotto lo stesso nome dove ogni variabile è un "cassetto" identificato da un numero. Ecco come si dichiara un array in C:

```
tipo nome_array[quantità];
```

Esempio:

```
int mio_array[10];
```

dichiara un array di 10 variabili int (N.B. da 0 a 9, non da 1 a 10!) chiamato mio_array. Se voglio cambiare un valore qualsiasi di questo array, basterà fare così:

```
mio_array[0] = 3;    // Il primo valore ora vale 3  
mio_array[1] = 2;    // Il secondo valore vale 2  
.....
```

Ovviamente posso anche leggere da tastiera il valore di un elemento dell'array:

```
printf ("Inserisci il valore del primo elemento: ");  
scanf("%d",&mio_array[0]);    // Leggo il valore del primo elemento  
  
printf ("Il primo elemento vale %d\n",mio_array[0]);
```

Posso anche leggere tutti i valori e poi stamparli tramite un ciclo for:

```
main() {  
    int mio_array[10];  
    int i;  
  
    for (i=0; i<10; i++) {                // Per i volte...
```

```

        printf ("Elemento n.%d: ",i);    // Elemento n.i
        scanf("%d",&mio_array[i]);    // Leggo un valore int dalla
tastiera                                // e lo memorizzo
nell'elemento numero                    // i dell'array.
    }

    for (i=0; i<10; i++)
        printf ("Elemento n.%d: %d\n",i,mio_array[i]); // Stampo
tutti i valori                            // contenuti
nell'array
}

```

Un array può anche essere dichiarato in modo esplicito con il suo contenuto:

```
int v[] = {2,4,6,2,6,5};
```

Vediamo ora un esempio più utile: un programma che calcola la media aritmetica di 5 numeri:

```

main() {
    float numeri[5];    // Array di 5 float
    float med=0;        // Media aritmetica
    int i;              // Variabile contatore

    for (i=0; i<5; i++) {
        printf ("Valore n.%d: ",i);
        scanf ("%f",&numeri[i]);
        med += numeri[i]; // Sommo fra loro tutti i numeri nell'array
    }

    med /= 5;          // Divido la somma dei numeri per la loro
quantità (5)

    printf ("Media aritmetica: %f\n",med);
}

```

Ancora un altro esempio, assimilabile all'algebra lineare vera e propria: un programmino che effettua il prodotto scalare tra due vettori (ricordo che dati due vettori v_1 e v_2 entrambi di n elementi il loro prodotto scalare è un numero uguale a $v_1[0]*v_2[0] + v_1[1]*v_2[1] + \dots + v_1[n-1]*v_2[n-1]$), dove gli elementi di entrambi i vettori sono stabiliti dall'utente via input:

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

```
// Dimensione dei due vettori
#define N 5
```

```
main() {
```

```

int v1[N],v2[N];
int i;
int prod=0;

for (i=0; i<N; i++) {
    printf ("Elemento %d del primo vettore: ",i+1);
    scanf ("%d",&v1[i]);
}

for (i=0; i<N; i++) {
    printf ("Elemento %d del secondo vettore: ",i+1);
    scanf ("%d",&v2[i]);
}

for (i=0; i<N; i++)
    prod += (v1[i]*v2[i]);

printf ("Prodotto scalare dei due vettori: %d\n", prod);
}

```

Matrici e array pluridimensionali

Negli esempi riportati sopra sono sempre presi in esame array a una dimensione, ovvero array dove ogni elemento è definito univocamente da un solo indice, che identifica la loro posizione all'interno dell'array stesso. Il C, al pari degli altri linguaggi di programmazione ad alto livello, mette anche a disposizione la possibilità di usare array a più dimensioni. Ci soffermeremo in particolar modo sugli array bidimensionali (dato che array di dimensioni superiori sono usati molto di rado), meglio conosciuti come *matrici*.

Una matrice si dichiara esattamente come un array monodimensionale, ma specificando sia il numero di righe che di colonne al suo interno:

```
int matrix[2][2]; // Dichiarata una matrice di interi 2x2
```

La lettura e la scrittura su questi elementi vengono effettuate in modo molto simile agli array, ma con due indici, in modo da gestire sia il numero di righe che di colonne della matrice:

```

int matrix[2][2];
int i,j;

// Leggo i valori della matrice da input
for (i=0; i<2; i++)
    for (j=0; j<2; j++) {
        printf ("Elemento [%d][%d]: ",i+1,j+1);
        scanf ("%d",&matrix[i][j]);
    }

// Stampo i valori della matrice

```

```
for (i=0; i<2; i++)
  for (j=0; j<2; j++)
    printf ("Elemento in posizione [%d][%d]: %d\n",i+1,j+1,matrix[i]
[j]);
```


I puntatori

.

La memoria RAM del calcolatore non è altro che un insieme di locazioni di memoria; per poter localizzare ciascuna locazione, ognuna di esse è identificata da un indirizzo univoco. Questo significa che:

- Per scrivere qualcosa in memoria centrale dobbiamo conoscere l'indirizzo del punto esatto in cui scrivere;
- Se conosciamo l'indirizzo di una data locazione possiamo leggere ciò che è contenuto al suo interno.

Strutture dinamiche

a differenza delle strutture dati di tipo statico che allocano i dati in "contenitori" prefissati (array, record etc etc), con i puntatori è possibile realizzare delle strutture dati DINAMICHE ottimizzando l'utilizzo della memoria e allo stesso tempo rendendo possibile l'implementazione di algoritmi che ottimizzano anche il tempo di esecuzione dell'elaborazione dei dati.

I puntatori sono il trampolino di lancio per la programmazione ad oggetti, infatti l'istanza di una classe non è altro che il puntatore ad un'area di memoria disegnata in maniera tale da corrispondere alla struttura della classe.

Liste monolanciate

[^start]-->[(DATA)(^next)]-->[(DATA)(^next)]-->[(DATA)(NULL)]

una lista è una struttura dati dinamica che a differenza dell'array non ha una lunghezza prestabilita, i suoi elementi formano una specie di "trenino" ovvero ognuno ha il collegamento all'elemento successivo e quindi il puntatore all'elemento successivo.

se l'elemento successivo non esiste (fine della lista) il collegamento conterrà il valore NULL ovvero non punterà a nulla.

questo tipo di lista è utilizzato per gestire le code dinamiche ed è possibile implementare diversi algoritmi, sia sequenziali che ricorsivi.

nelle strutture a Lista è importante mantenere sempre il puntatore(testa) al primo elemento della lista (nell'esempio ^start) se si perde quel riferimento, la lista andrà

persa.

supponiamo di essere un elemento che si vuole inserire nella lista.

- se vogliamo inserirci alla fine basterà allocarci da qualsiasi parte sulla ram, quindi modificare il puntatore next dell'ultimo elemento della lista in modo tale che punti a noi.

- se vogliamo inserirci all'inizio invece dobbiamo FARCI UNA COPIA del puntatore di testa se no perdiamo il riferimento alla lista, quindi modificare la testa in modo tale che punti a noi, e noi punteremo all'elemento precedentemente puntato dalla testa.

la lista è monolanciata, quindi è possibile scorrerla in un solo senso, supponiamo ad esempio di fermarci al secondo elemento, e di volerci inserire tra il primo e il secondo, non possiamo modificare il puntatore del primo elemento, almeno che non scorriamo di nuovo tutta la lista fino ad arrivare al primo elemento.

per ottimizzare il tutto basterà inserirci dopo il secondo elemento (quindi tra il secondo ed il terzo, che è fattibile senza dover ripercorrere la lista) e quindi invertire il nostro contenuto informativo con quello del secondo elemento.

Liste circolari

implementate negli scheduler, le liste circolari non terminano con NULL ma con il puntatore al primo elemento, in questo modo è possibile percorrerle costantemente. supponiamo ad esempio che ogni elemento sia un'operazione da compiere

il processo E scorrerà in sequenza la lista, eseguendo l'operazione che trova il processo M modificherà invece i collegamenti, ed eventualmente inserirà nuove operazioni per modificare il percorso del processo E.

Alberi e Grafi

le strutture dati consentono di modellizzare la realtà, a seconda della complessità di un fenomeno fisico, esso può essere rappresentato con strutture lineari come gli array oppure con strutture dinamiche o ad oggetti.

La Vita non può essere rappresentata con strutture dati lineari come gli array, altrimenti potrebbe essere zippata.. la Vita è sinonimo di Infinito, spazia in ogni dimensione e quindi non può essere modellizzata con una strutture dati e tantomeno compressa, poichè ogni spetto, anche il più piccolo è una parte di Infinito.

Nell'esempio precedente possiamo immaginare il processo E come un treno che si muove ed il processo M come il gestore del percorso che lo dirige.

le strutture ad Albero ed i grafi consentono di rappresentare la realtà ottimizzando

ulteriormente la modellizzazione.

un grafo è un insieme di nodi da cui dipartono dei grafi è quindi una struttura ricorsiva e può essere immaginata come un albero fatto di nodi e di rami da ogni ramo si collega ad un nodo da cui dipartono altri rami, e così via.

Pensiamo ad esempio ad Internet, Internet è un grafo, gestito da degli algoritmi che lavorano sui grafi, come il famoso algoritmo di Dijkstra.

gli alberi sono un particolare tipo di grafo in cui due rami non possono puntare allo stesso nodo, e tra questi si distinguono gli alberi bilanciati nei quali da un nodo dipartono solo due rami ed il numero di nodi e sottonodi presenti in una diramazione devono corrispondere a quelli presenti nell'altra diramazione.

Questa struttura dati è particolarmente indicata per la ricerca dicotomica.

Puntatori in C

Il C consente di gestire, oltre al contenuto delle variabili stesse, anche i loro indirizzi (ovvero le loro locazioni in memoria) attraverso il meccanismo dei puntatori.

Fino ad oggi abbiamo gestito le variabili all'interno dei blocchi di codice in cui tali variabili erano visibili, quindi non c'è stata la reale necessità di utilizzare l'indirizzo della locazione di memoria in cui i valori di tali variabili erano stati memorizzati. L'uso però a volte diventa indispensabile all'interno delle funzioni (anche nella scanf, come avevo anticipato, si usava implicitamente un puntatore per stabilire l'area fisica di memoria in cui salvare la variabile letta da input).

Un puntatore ha una sintassi simile:

```
int a=3; // Variabile
int *x; // 'Puntatore' ad una variabile di tipo int

x=&a; // Il puntatore 'x' contiene l'indirizzo della variabile
      'a'
*x=4; // In questo modo modifico il contenuto del valore puntato da
x,
      // quindi modifico indirettamente il valore di a
```

&a identifica l'indirizzo in memoria al quale si trova la variabile a, indirizzo che viene salvato nel puntatore x. Quest'uso dei puntatori dovrebbe farci tornare alla mente la sintassi della scanf:

```
int a;

printf ("Inserisci un valore intero: ");
scanf ("%d",&a);
```

Ora possiamo capire appieno la sintassi della scanf. È una funzione che non fa altro che leggere, in questo caso, un valore intero da tastiera, e salvarlo nell'indirizzo fisico

di memoria in cui si trova la variabile a ($\&a$).

Ovviamente, se voglio salvare un valore letto da tastiera in una variabile a cui è già associato un puntatore, non ho bisogno di ricorrere alla scrittura di sopra:

```
#include <stdio.h>

main() {
    int a;
    int *x=&a;

    printf ("Inserisci un valore intero: ");

    // Salvo il valore immesso direttamente nell'allocazione
    // di memoria puntata da 'x', ovvero nella variabile 'a'
    scanf ("%d",x);

    printf ("Valore salvato all'indirizzo: 0x%x: %d\n",x,a);
}
```

ritornerà come output qualcosa del tipo

```
Valore salvato all'indirizzo: 0xbfc16a24: 4
```

dove 0xbfc16a24 è, in questo caso, l'indirizzo fisico di memoria (in formato esadecimale) in cui si trova la variabile intera a (e quindi il valore 4, in questo caso). Ricordiamo che sulle macchine a 32 bit (quindi tutte le macchine Intel dal 486 in su, escluse quelle a 64 bit come Itanium e simili) un indirizzo di memoria è *sempre* grande 32 bit (come in questo caso), quindi in memoria un puntatore occuperà sempre, indipendentemente dalla variabile a cui punta, 32 bit = 4 byte.

Passaggio di puntatori alle funzioni

Vediamo ora un'applicazione pratica dell'uso dei puntatori. Abbiamo una classica applicazione che effettua lo scambio di due numeri interi (ovvero, se ho due variabili, $a=3$ e $b=4$, voglio ottenere $a=4$ e $b=3$). Il modo più immediato di risolvere questo problema è quello di appoggiarsi ad una variabile temporanea:

```
int a=4;
int b=3;
int tmp;

.....

tmp=a;    // tmp=4
a=b;     // a=3
b=tmp    // b=tmp=4
```

Vogliamo ora implementare questo codice in una funzione a parte che viene poi richiamata dal nostro programma. Con le nostre conoscenze attuali scriveremo un

codice del genere:

```
#include <stdio.h>

// Funzione per lo scambio
void swap(int a, int b) {
    int tmp;

    tmp=a;
    a=b;
    b=tmp;
}

main() {
    int a=4;
    int b=3;

    printf ("a=%d, b=%d\n",a,b);
    swap(a,b);
    printf ("a=%d, b=%d\n",a,b);
}
```

compilandolo avremo una sorpresa inaspettata: i valori sono rimasti immutati. Questo perché alla funzione swap non passiamo le variabili *fisicamente*, ma passiamo i loro valori. Quando invociamo una funzione, l'atto della chiamata crea in memoria una nuova area dello stack associata alla funzione appena chiamata. In questo stack vengono *copiati* i valori degli argomenti passati. La funzione quindi non opera fisicamente sulle variabili passate, ma opera piuttosto su *copie* di esse. Quando la funzione termina l'area dello stack corrispondente viene distrutta, e con essa anche le copie delle variabili al suo interno, quindi non è possibile tener traccia delle modifiche.

La soluzione è proprio quella di *ricorrere ai puntatori*, ovvero non passare alla funzione copie delle variabili, ma gli indirizzi fisici in cui esse si trovano, in modo che la funzione agirà direttamente su quegli indirizzi:

```
#include <stdio.h>

// Funzione per lo scambio
void swap(int *a, int *b) {
    int tmp;

    tmp=*a;    // tmp contiene il contenuto della variabile intera
puntata da a
    *a=*b;    // a contiene il contenuto della variabile intera
puntata da b
    *b=tmp;    // b contiene il contenuto della variabile intera
puntata da tmp
}

main() {
    int a=4;
```

```

int b=3;

printf ("a=%d, b=%d\n",a,b);

// Non passo le variabili alla funzione ma i loro indirizzi in
memoria
swap(&a,&b);

printf ("a=%d, b=%d\n",a,b);
}

```

e ora il nostro codice funziona a dovere.

Puntatori e array

Nel paragrafo precedente abbiamo visto gli array come oggetti a sé stanti, diversi da qualsiasi altro tipo di variabile e di dato che abbiamo incontrato. Ai fini del calcolatore però un array viene trattato esattamente alla stregua di un puntatore, un puntatore all'area di memoria dov'è contenuto il primo elemento dell'array stesso. Esempio:

```

#include <stdio.h>

main() {
    int v[] = {4,2,8,5,2};

    // Queste due scritture sono equivalenti
    printf ("Primo elemento dell'array: %d\n",v[0]);
    printf ("Primo elemento dell'array: %d\n",*v);
}

```

questo vuol dire che possiamo accedere a qualsiasi elemento dell'array specificando o il suo indice tra parentesi quadre o sommandolo al valore del puntatore al primo elemento:

```

#include <stdio.h>

main() {
    int v[] = {4,2,8,5,2};

    // Queste due scritture sono equivalenti
    printf ("Secondo elemento dell'array: %d\n",v[1]);
    printf ("Secondo elemento dell'array: %d\n",*(v+1));
}

```

questo perché quando viene istanziato un array non viene fatto altro che creare un puntatore ad una certa area della memoria centrale, per poi riservare tanto spazio in memoria quanto specificato dalla dimensione dell'array (nell'esempio di sopra lo spazio di 5 variabili int, una variabile int in genere è grande 4 byte su una macchina a 32 bit quindi vengono riservati $5*4=20$ byte a partire dall'indirizzo del primo

elemento).

Passaggio di array a funzioni

Tale caratteristica si rivela conveniente per molti aspetti. L'aspetto principale consiste nel poter passare un array ad una funzione come se fosse un puntatore. Esempio:

```
#include <stdio.h>

int print_array (int *v, int dim) {
    int i;

    for (i=0; i<dim; i++)
        printf ("Elemento [%d]: %d\n",i,v[i]);
}

main() {
    int v[] = { 3,5,2,7,4,2,7 };

    print_array(v,7);
}
```

Allocazione dinamica della memoria

L'altro enorme vantaggio di quest'ottica da parte del C (ovvero il considerare gli array come semplici puntatori) risiede nel poter allocare dinamicamente dello spazio in memoria. Non sempre sappiamo a priori quanto spazio può servire in memoria per un array usato nel nostro programma. Ad esempio, nel caso in cui si dà la possibilità all'utente di stabilire il numero di elementi da inserire o quando vogliamo salvare dei dati in memoria senza sapere ancora la quantità di dati da salvare (in questo caso dovremmo prima contare il numero di dati da salvare, quindi allocare tanta memoria da poterli mantenere). In questi casi ci viene in aiuto una delle caratteristiche più potenti del C, l'allocazione dinamica della memoria, allocazione che è possibile attraverso la funzione *malloc*, definita in *stdlib.h*. La *malloc* ha una sintassi simile:

```
void malloc (int size);
```

al posto di *size* specificheremo quanta memoria vogliamo allocare per la nostra variabile o il nostro array. Come è possibile vedere il valore di ritorno di questa funzione è *void*, ovvero ritorna l'indirizzo della zona di memoria allocata in formato 'grezzo'. Per questo motivo è necessario *specializzare* la funzione attraverso un operatore di cast. Esempio chiarificatore:

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

main() {
    int *v;
    int i,n;
```

```

printf ("Quanti elementi vuoi inserire nell'array? ");
scanf ("%d",&n);

v = (int*) malloc(n*sizeof(int));

for (i=0; i<n; i++) {
    printf ("Elemento n.%d: ",i+1);
    scanf ("%d",&v[i]);
}

for (i=0; i<n; i++)
    printf ("Elemento n.%d: %d\n",i+1,v[i]);
}

```

La scrittura *sizeof(int)* ritorna la dimensione di una variabile int sulla macchina in uso, quindi *n*sizeof(int)* è il numero di byte effettivi da allocare in memoria (ovvero nella malloc diciamo di allocare in memoria n blocchi di dimensione *sizeof(int)* l'uno che ospiteranno n variabili intere, e salviamo l'indirizzo a cui comincia questa zona di memoria nel puntatore v).

Ciò è possibile anche con array di dimensione superiore a 1 (es. per le matrici):

```

#include <stdio.h>
#include <stdlib.h>

main() {
    int **m;
    int i,j;
    int m,n;

    printf ("Numero di righe della matrice: ");
    scanf ("%d",&m);

    printf ("Numero di colonne della matrice: ");
    scanf ("%d",&n);

    m = (int**) malloc(m*n*sizeof(int));

    // Inizializzo anche tutti i sotto-vettori, ovvero le righe della
matrice
    for (i=0; i<m; i++)
        m[i] = (int*) malloc(n*sizeof(int));

    for (i=0; i<m; i++)
        for (j=0; j<n; j++) {
            printf ("Elemento [%d][%d]: ",i+1,j+1);
            scanf ("%d",&v[i][j]);
        }

    for (i=0; i<m; i++)
        for (j=0; j<n; j++)

```



```
    printf ("Elemento [%d][%d]: %d\n",i+1,j+1,v[i]);  
}
```

Puntatori a funzioni

Le funzioni a basso livello non sono altro che sequenze di istruzioni binarie piazzate nella memoria centrale, al pari di una qualsiasi variabile. È quindi possibile anche costruire puntatori che puntino a funzioni, in quanto normali aree di memoria. La sintassi è la seguente:

*tipo (*nome_ptr)(argomenti) = funzione*

Per richiamare la funzione puntata, basta poi un

*(*nome_ptr)(argomenti)*

Esempio:

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

```
void foo() {  
    printf ("Ciao\n");  
}
```

```
main() {  
    void (*ptr)(void) = foo;  
    printf ("foo si trova all'indirizzo 0x%.8x\n",ptr);  
    (*ptr)();  
}
```

o ancora

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

```
int foo(int a, int b) {  
    return a+b;  
}
```

```
main() {  
    int a=2,b=3;  
    int (*ptr)(int, int) = foo;  
    printf ("foo si trova all'indirizzo 0x%.8x\n",ptr);  
    printf ("%d+%d=%d\n",a,b,(*ptr)(a,b));  
}
```

Funzioni di callback

La conseguenza immediata dei puntatori a funzione sono le *funzioni di callback*. Ovvero, nulla mi impedisce, a questo punto, di passare come parametro di una funzione un puntatore a funzione, e la funzione richiamata può richiamare la

funzione passata come argomento. Esempio:

```
#include <stdio.h>

void print () {
    printf ("Ciao\n");
}

int foo(void (*f)(void)) {
    f();
}

main() {
    foo(print);
}
```

Stringhe

.

La gestione delle stringhe è alla base della programmazione in qualsiasi linguaggio di programmazione. Ogni oggetto che viene stampato sullo schermo è una stringa. I messaggi che abbiamo scritto finora su stdout con la printf non sono altro che stringhe, lo stesso vale anche per le stringhe di formato della scanf ecc.

In C una stringa non è altro che un array di elementi di tipo char. Linguaggi di programmazione più moderni, come Java, Perl, Python, PHP e lo stesso C++, tramite l'uso della classe 'string', consentono di usare le stringhe in modo più avanzato, come tipi predefiniti all'interno del linguaggio stesso. La visione del C (ovvero stringhe=array di tipo char) può essere più macchinosa e a volte anche più pericolosa, ma mette in mano al programmatore la gestione di questo tipo di entità al 100%.

Dichiarazione di una stringa

Abbiamo visto che in C una stringa non è altro che un array di elementi di tipo char. Questo ci fa pensare subito a un tipo di dichiarazione immediato (ma alquanto scomodo):

```
char my_string[] = { 'H', 'e', 'l', 'l', 'o' };
```

La dichiarazione vista sopra non è comodissima, ragion per cui il C consente di dichiarare le stringhe direttamente così:

```
char my_string[] = "Hello";
```

o ancora così, sfruttando una scrittura di tipo puntatore:

```
char *my_string = "Hello";
```

Ovviamente possiamo anche dichiarare delle stringhe senza inizializzarle. In questo caso le dichiariamo specificando il nome e la dimensione:

```
char my_string[20]; // Stringa che può contenere 20 caratteri
```

e vale anche lo stesso discorso che abbiamo fatto con gli array per l'inizializzazione dinamica di una stringa:

```
char *my_string;  
int n;
```

.....

```
printf ("Quanti caratteri deve contenere la tua stringa? ");
scanf ("%d",&n);
```

```
my_string = (char*) malloc (n*sizeof(char));
```

Per leggere una stringa invece possiamo ricorrere alla funzione scanf, passando come stringa di formato '%s':

```
char str[20];
```

```
.....
```

```
printf ("Inserisci una stringa: ");
scanf ("%s",str);
```

Si noti che non ho usato la scrittura '&str' nella scanf, in quanto la stringa già di suo rappresenta un puntatore (in quanto un array non è altro che, a livello del compilatore, un puntatore al suo primo elemento, come abbiamo visto prima).

Attenzione: l'uso della scanf per la lettura delle stringhe è potenzialmente dannoso per la stabilità e la sicurezza di un programma. In seguito valuteremo metodi per fare letture in tutta sicurezza.

Esercizio pratico: un programmino che prende in input una stringa e trasforma tutti i suoi eventuali caratteri alfabetici maiuscoli in caratteri minuscoli:

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

// Funzione per la conversione di tutti i caratteri
// maiuscoli in caratteri minuscoli
void toLower(char *s) {
    int i;

    for (i=0; i<strlen(s); i++)
        // Se il carattere corrispondente della stringa è
        // un carattere maiuscolo, ovvero è compreso tra A e Z...
        if ( (s[i]>='A') && (s[i]<='Z') )
            s[i]+=32;
}

main() {
    char s[20];
    int i;

    printf ("Inserisci una stringa: ");
    scanf ("%s",s);

    toLower(s);
    printf ("Stringa convertita completamente in caratteri
minuscoli: %s\n",s);
```

```
}
```

Da notare l'uso della funzione `strlen`, definita in `string.h`. Tale funzione ritorna la lunghezza di una stringa, ovvero il numero di caratteri presenti fino al carattere terminatore della stringa. Ogni stringa possiede infatti un carattere terminatore per identificarne la fine (ovvero fin dove il compilatore deve leggere il contenuto della stringa). Tale carattere è, per convenzione, il carattere NULL, identificato dalla sequenza di escape `'\0'` e associato al codice ASCII 0. Ogni stringa quindi, anche se non è specificato, ha N+1 caratteri, ovvero gli N caratteri che effettivamente la compongono e il carattere NULL che ne identifica la fine:

```
char *str = "Hello";  
// In realtà a livello del compilatore 'str' è vista come  
// 'H','e','l','l','o','\0'
```

Con le conoscenze che abbiamo in questo momento possiamo anche capire come è scritto il codice della funzione `strlen`:

```
int strlen(char *s) {  
    int len;  
  
    for (len=0; s[len]!='\0'; len++);  
    return len;  
}
```

ovvero un ciclo `for` dove la variabile contatore viene incrementata finché il carattere corrispondente all'indice all'interno della stringa non è uguale al carattere NULL (appunto con codice ASCII uguale a 0). Il valore della variabile contatore a questo punto rappresenta il numero effettivo di caratteri fino al NULL, ovvero il numero effettivo di caratteri all'interno della stringa, valore che viene ritornato dalla funzione. Scrivere un

```
for (i=0; i<strlen(s); i++)
```

equivale quindi a dire "cicla finché la stringa `s` contiene dei caratteri, o finché non viene raggiunta la fine della stringa".

Questa scrittura

```
if ( (s[i]>='A') && (s[i]<='Z') )  
    s[i]+=32;
```

equivale a dire "se il carattere attuale è maggiore o uguale ad A e minore o uguale a Z, ovvero è una lettera maiuscola, somma al suo valore ASCII attuale il valore 32". 32 è l'offset che nella tabella dei caratteri ASCII esiste tra i caratteri maiuscoli e quelli minuscoli. Per verificare:

```
printf ("%d\n", 'a' - 'A');
```

Operare sulle stringhe - La libreria string.h

Abbiamo incontrato nel paragrafo precedente la funzione *strlen*, definita in *string.h*. Questo header mette a disposizione molte funzioni per operare su questi tipi di dati. Tenteremo di esaminare le più importanti nel corso di questo paragrafo.

strcmp

La funzione *strcmp* (STRing CoMPare) confronta tra di loro i valori di due stringhe, il suo prototipo è qualcosa di simile:

```
int strcmp(const char *s1, const char *s2);
```

dove *s1* e *s2* sono le due stringhe da confrontare. La funzione ritorna

- Un valore > 0 se da un confronto byte a byte *s1* ha più caratteri il cui codice ASCII è maggiore del corrispondente codice ASCII di *s2*
- 0 se le due stringhe sono uguali
- Un valore < 0 nei casi rimanenti

questa funzione è utilizzatissima per vedere se due stringhe hanno lo stesso contenuto. Sono infatti da evitare come la peste scritte del genere:

```
char *s1;  
char *s2="pippo";
```

```
.....
```

```
if (s1==s2)  
    printf ("Ciao pippo\n");
```

questo perché la scrittura sopra non fa altro che confrontare gli indirizzi fisici in memoria delle due stringhe, ed effettuare le operazioni richieste se gli indirizzi coincidono. Ciò ovviamente non sarà mai verificato, dato che due variabili diverse in memoria hanno anche indirizzi diversi, quindi il codice scritto sopra è praticamente inefficiente. Per confrontare due stringhe è invece necessario ricorrere alla funzione *strcmp*, ricordando che la funzione ritorna 0 quando il contenuto di due stringhe è lo stesso. Ecco quindi la versione corretta del codice di sopra:

```
char *s1;  
char *s2="pippo";
```

```
.....
```

```
if (!strcmp(s1,s2))  
    // Equivale a scrivere  
    // if (strcmp(s1,s2)==0)  
    printf ("Ciao pippo\n");
```

strncmp

La funzione *strncmp* è molto simile a *strcmp*, con l'eccezione che confronta solo i primi *n* caratteri sia di *s1* che di *s2*. La sua sintassi è qualcosa di simile:

```
int strncmp(const char *s1, const char *s2, size_t n);
```

dove *n* è il numero di caratteri da confrontare.

strcpy

La funzione *strcpy* copia una stringa in un'altra. La sua sintassi è qualcosa di simile:

```
char *strcpy(char *dest, const char *src);
```

dove *dest* è la stringa all'interno della quale viene copiato il nuovo valore e *src* è la stringa da copiare. Il valore di ritorno della funzione è un puntatore a *dest*.

Quando si vuole copiare una stringa in un'altra è infatti sconsigliabile usare una scrittura del genere:

```
char *s1="pippo";  
char *s2;
```

```
s2=s1; // ATTENZIONE!!
```

La scrittura di sopra infatti copia il puntatore al primo elemento della stringa *s1* nella stringa *s2*. Ciò vuol dire che ogni eventuale modifica di *s1* modifica anche *s2*, dato che entrambe le variabili agiscono sulla stessa zona di memoria, e viceversa, il che è decisamente un effetto collaterale. La scrittura corretta è qualcosa del tipo

```
char *s1="pippo";  
char s2[32];
```

```
strcpy(s2,s1);
```

in quanto la funzione *strcpy* genera in *s2* una copia esatta di *s1*, che però, essendo residente in una zona di memoria diversa, è completamente indipendente da *s1*.

La funzione *strcpy* ha un codice simile:

```
char* strcpy (char *s1, char *s2) {  
    int i;  
  
    // Finché la stringa s2 ha dei caratteri...  
    for (i=0; i<strlen(s2); i++)  
        // ...copia il carattere in s1  
        s1[i]=s2[i];  
  
    return s1;  
}
```

Questa funzione è però potenzialmente dannosa per la sicurezza dell'applicazione.

Vedi il paragrafo su "Uso delle stringhe e sicurezza del programma".

strncpy

La funzione *strncpy* ha una sintassi molto simile a *strcpy*, con la differenza che copia solo i primi *n* caratteri della stringa sorgente nella stringa di destinazione, per tenere il processo di copia sotto controllo ed evitare problemi di sicurezza nell'operazione, come vedremo in seguito. La sua sintassi è

```
char *strncpy(char *dest, const char *src, int n);
```

La sintassi è la stessa di *strcpy*, a parte per *n*, che identifica appunto il numero di caratteri di *src* che verranno copiati in *dest*. L'uso di questa funzione è preferibile a quello di *strcpy* quando possibile, proprio per evitare problemi di sicurezza legati ad una copia non controllata.

strcat

La funzione *strcat* concatena l'inizio di una stringa alla fine di un'altra. La sua sintassi è

```
char *strcat(char *dest, const char *src);
```

dove *dest* è la stringa alla cui fine viene concatenata la stringa *src*. Esempio di utilizzo:

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

main() {
    char s1[20];
    char *s2 = "pippo";

    // Copio all'interno di s1 la stringa "Ciao "
    // copiando esattamente il numero di byte che
    // mi servono, tramite l'operatore sizeof
    strncpy (s1,"Ciao ",sizeof("Ciao "));

    strcat (s1,s2);
    // s1 ora contiene "Ciao pippo"
}
```

La funzione *strcat* ritorna un puntatore a *char* che rappresenta un puntatore alla zona di memoria dove è salvato *dest*.

Attenzione: anche la *strcat*, così come la *strcpy* ed altre funzioni che vedremo in seguito, è sul libro nero delle funzioni a potenziale rischio di sicurezza per un'applicazione. Il suo uso, quando possibile, va evitato.

strncat

La sua sintassi è molto simile a `strcat`, con la differenza che in `strncat` vanno specificati anche il numero di caratteri di `src` da copiare in `dest`, in modo da tenere la copia sotto controllo. La sua sintassi è

```
char *strncat(char *dest, const char *src, int n);
```

dove `n` rappresenta il numero di caratteri di `src` da copiare. Il suo uso, quando possibile, è preferibile a quello di `strcat`.

strstr

La funzione `strstr` serve per verificare se esiste una sottostringa all'interno della stringa di partenza. La sua sintassi è

```
char *strstr(const char *haystack, const char *needle);
```

dove `haystack` (lett. 'pagliaio') è la stringa all'interno della quale cercare, `needle` (lett. 'ago') è la stringa da cercare (da notare ancora una volta il sottile umorismo degli sviluppatori del C). La funzione ritorna

- Un puntatore intero, che rappresenta la zona di memoria in cui è stata trovata la sottostringa, nel caso in cui la sottostringa dovesse essere trovata
- NULL nel caso in cui la sottostringa non dovesse essere trovata

Esempio:

```
/*
 * Questo programmino chiede in input all'utente due stringhe e
 * verifica se la seconda stringa è localizzata all'interno della
 * prima
 */

#include <stdio.h>
#include <string.h>

main() {
    char s1[32];
    char s2[32];

    printf ("Inserire la stringa all'interno della quale cercare: ");
    scanf ("%s",s1);

    printf ("Inserire la stringa da cercare: ");
    scanf ("%s",s2);

    if (strstr(s1,s2))
        // Equivale a scrivere
        // if (strstr(s1,s2) != 0)
        // ovvero se il valore di ritorno della funzione non è NULL
        printf ("Stringa \"%s\" trovata all'interno di \"%s\", in
posizione %d\n",
```

```

        s2,s1,(strstr(s1,s2)-s1));
    else
        printf ("Stringa  \"%s\"  non trovata all'interno
di \"%s\"\n",s2,s1);
}

```

Si noti questa scrittura:

```
strstr(s1,s2)-s1
```

`strstr` ritorna infatti l'indirizzo dell'area di memoria in cui si trova `s2` all'interno di `s1`. Se a questo indirizzo sottraggo l'indirizzo di `s1`, ovvero l'indirizzo del primo carattere di `s1`, ottengo la locazione effettiva della sottostringa all'interno della stringa di partenza. Se ad esempio `s1="Ciao pippo"` e `s2="pippo"`, `strstr(s1,s2)-s1 = 5`.

Altre funzioni sulle stringhe

sprintf

La funzione *sprintf*, definita in `stdio.h`, è del tutto analoga alla `printf`. La differenza è che la `printf` scrive dell'output formattato su standard output, mentre la `sprintf` scrive dell'output formattato direttamente su una stringa, che rappresenta il primo argomento della funzione. Esempio:

```

#include <stdio.h>

main() {
    char s1[32];
    char s2="Ciao";
    char s3="pippo";
    int age=2;

    sprintf (s1,"%s %s ho %d anni",s2,s3,age);
    // Scrivo su s1 attraverso la sprintf
    // Ora s1 contiene la stringa "Ciao pippo ho 2 anni"
}

```

Anche la funzione `sprintf` è sulla lista di quelle da usare con cautela, e solo quando si è sicuri che la stringa di destinazione è in grado di contenere tutti i byte che si stanno per copiare al suo interno.

snprintf

La funzione *snprintf* è un'alternativa più sicura alla `sprintf`, e al suo interno va specificato anche il numero massimo di caratteri da copiare. La sua sintassi è quindi

```
int snprintf(char *str, int size, const char *format, ...);
```

dove *size* rappresenta il numero massimo di byte della stringa di formato da copiare

all'interno di *str*.

sscanf

La funzione *sscanf* è del tutto analoga alla *scanf* classica, solo che invece di leggere i dati dalla tastiera li legge dall'interno di una stringa. Esempio, ecco un uso classico di *sscanf*. Abbiamo una stringa che rappresenta una data, in formato 'gg/mm/aaaa'. Vogliamo ottenere, dall'interno di questa stringa, il giorno, il mese e l'anno e salvarli all'interno di 3 variabili intere. Con *sscanf* la cosa è presto fatta:

```
char *date = "13/08/2007";
int d,m,y;

sscanf (date,"%d/%d/%d",&d,&m,&y);
// d=13, m=8, y=2007
```

La funzione ritorna un intero che rappresenta il numero di campi letti all'interno della stringa. Il controllo su questo valore di ritorno può tornare utile per verificare se l'utente ha inserito la stringa nel formato giusto:

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

main() {
    char date[16];
    int d,m,y;

    printf ("Inserisci una data: ");
    scanf ("%s",date);

    // Se non leggo almeno 3 interi nella stringa
    // inserita separati da '/'...
    if (sscanf(date,"%d/%d/%d",&d,&m,&y) != 3) {
        printf ("Errore: data inserita non valida: %s\n",date);
        exit(1);
    }

    printf ("Giorno: %d\n",d);
    printf ("Mese: %d\n",m);
    printf ("Anno: %d\n",y);
}
```

gets

Un piccolo limite della lettura delle stringhe sta nel fatto che la lettura si interrompe quando incontra uno spazio. Se ad esempio un'applicazione richiede all'utente di inserire una stringa e l'utente inserisce "Ciao mondo", se la lettura avviene tramite *scanf* molto probabilmente la stringa risultante dopo la lettura sarà semplicemente "Ciao". Per evitare questo piccolo inconveniente si ricorre alla funzione *gets*,

nonostante il suo uso sia deprecato dai nuovi standard C. Esempio di utilizzo:

```
#include <stdio.h>

main() {
    char str[32];

    printf ("Inserisci una stringa: ");
    gets (str);

    // Se ora inserisco stringhe con degli spazi in mezzo vengono
    // salvate ugualmente nella
    // stringa finale, in quanto la gets legge tutti i caratteri fino
    // al fine linea

    printf ("Stringa inserita: %s\n",str);
}
```

Attenzione: anche la `gets` è nella lista delle funzioni a rischio. Presto vedremo subito in che modo operare sulle stringhe con le funzioni giuste e nel modo giusto, e i rischi che si ottengono lavorando in modo non controllato sulle stringhe.

atoi

La funzione `atoi` (ASCII to int), definita in `stdlib.h`, è usata per convertire una stringa in un valore intero. La sua sintassi è molto semplice:

```
int atoi(const char *nptr);
```

e restituisce il valore convertito. Nel caso in cui la stringa non contenga un valore numerico valido, la funzione ritorna zero. Esempio di utilizzo:

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

main() {
    int n;
    char *s = "3";

    n=atoi(s);
    // Ora n contiene il valore intero 3
}
```

Della stessa famiglia sono le funzioni `atol` (ASCII to long) e `atof` (ASCII to float).

Argomenti passati al main

Sappiamo che molte applicazioni accettano una lista di parametri passati dall'utente in input. Ad esempio, il comando `dir` del DOS è in grado di accettare alcuni parametri per configurarne il funzionamento (ad esempio `dir /h` e simili...). Idem per `net send`

(net send indirizzo_host messaggio) e per, ad esempio, il comando ls di Unix (ls -l -h). È possibile passare questi parametri ad un programma sfruttando gli argomenti del main. Il main è una funzione come tutte le altre, e quindi può anche ricevere argomenti in input. In particolare, è possibile leggere i parametri eventuali passati ad un programma tramite l'uso di *argv*, un vettore di stringhe da passare al main. Il numero di argomenti passati viene invece salvato nella variabile intera *argc*. Esempio di utilizzo:

```
#include <stdio.h>

main(int argc, char **argv) {
    printf ("Nome dell'eseguibile in esecuzione: %s\n",argv[0]);
}
```

La prima stringa del vettore *argv* contiene infatti il nome dell'eseguibile (quindi la variabile *argc* è sempre settata almeno a uno). Gli eventuali argomenti successivi passati al programma vengono salvati in *argv[1],...,argv[n]*. Esempio pratico:

```
#include <stdio.h>

main(int argc, char **argv) {
    int i;

    printf ("Argomenti passati al programma:\n");

    for (i=1; i<argc; i++)
        printf ("%s\n",argv[i]);
}
```

Se compiliamo questo eseguibile come 'stampa_arg' e lo invochiamo con gli argomenti "Ciao mondo come stai", così (in ambiente Unix):

```
./stampa_arg Ciao mondo come stai
```

avremo come output qualcosa del tipo

```
Argomenti passati al programma: Ciao mondo come stai
```

Uso delle stringhe e sicurezza del programma

Nei paragrafi precedenti abbiamo preso in esame alcune funzioni sulle stringhe che possono rivelarsi potenzialmente dannose per la stabilità e la sicurezza di un'applicazione. In questo paragrafo esamineremo i rischi concreti connessi ad una cattiva gestione delle stringhe.

Esempio pratico:

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

main() {
```

```

char s1[2];
char *s2 = "La vista teresa sta tra l'erbetta";

strcpy (s1,s2);
}

```

Eseguendo un codice del genere molto probabilmente l'applicazione andrà in crash. Se siamo su un sistema Unix il kernel ci risponderà con un bel segmentation fault, su Windows ci comparirà una finestra che ci avverte che l'applicazione ha tentato la scrittura su un'area di memoria non valida. Quello che abbiamo fatto è tentare di copiare in un buffer più byte di quelli che il buffer stesso può contenere, e in modo non controllato (la strcpy non effettua una copia controllata, non si ferma se i limiti di capienza della stringa vengono raggiunti). Il risultato è che l'applicazione va in crash, in quanto, attraverso la strcpy, è andata a scrivere su una zona di memoria al di fuori di quella della stringa stessa, andando a sovrascrivere l'indirizzo di ritorno della funzione con un indirizzo non valido. L'indirizzo viene letto dalla CPU, che tenta di leggere l'istruzione a quell'indirizzo. Indirizzo che nella maggior parte dei casi non sarà un indirizzo di memoria valido, quindi provocherà il crash del programma.

Ma il crash del programma, nonostante sia un danno non da poco, non è nemmeno il minore dei danni. Esempio pratico con un'applicazione:

```

#include <stdio.h>
#include <string.h>

main(int argc, char **argv) {
    char str[16];

    strcpy (str,argv[1]);
}

```

Attenzione all'uso di strcpy in questa applicazione. La funzione copia il primo argomento passato al programma nella stringa str, che può tenere 16 caratteri, senza fare ulteriori controlli sulla lunghezza effettiva della stringa da copiare. Proviamo ad avviare l'applicazione con il nostro debugger preferito (in questo caso userò Gdb) per vedere cosa succede in memoria quando passo al programma un argomento molto lungo:

```

(gdb)      run      `perl      -e      'print      "A"      x32`
Starting program: /home/blacklight/prog/c/5 `perl -e 'print "A" x32`
Program      received      signal      SIGSEGV,      Segmentation      fault.
0x41414141 in ?? ()

```

Il comando ``perl -e 'print "A" x32`` non fa altro che richiamare l'interprete Perl (un linguaggio di programmazione), stampando la lettera "A" 32 volte (un modo per evitare di scrivere 32 volte "A", giusto una comodità). Il programma, tentando di copiare un buffer troppo grande in una stringa che non è in grado di contenere tanti byte, va in crash. Ma vediamo cosa succede a livello dei registri:

```

(gdb)
                                i
                                r
eax          0xbfab7890          -1079281520
ecx          0xfffff098          -3944
edx          0xbfab8818          -1079277544
ebx          0xb7edefc           -1209143300
esp          0xbfab78b0          0xbfab78b0
ebp          0x41414141          0x41414141
esi          0xbfab7934          -1079281356
edi          0xbfab78c0          -1079281472
eip          0x41414141          0x41414141
eflags      0x210282          [ SF IF RF ID ]
cs          0x73                115
ss          0x7b                123
ds          0x7b                123
es          0x7b                123
fs          0x0                  0
gs 0x33 51

```

Da notare il registro EIP. Tale registro contiene, nelle architetture Intel-based, l'indirizzo in memoria della prossima istruzione da eseguire. L'indirizzo è stato sovrascritto da una sequenza di 0x41. E 0x41, in esadecimale, corrisponde al carattere ASCII "A". In pratica il nostro buffer lungo è andato a sovrascrivere il registro EIP, cambiando il valore dell'indirizzo della prossima istruzione da pescare in memoria. In questo caso, la sequenza di 0x41 non rappresenta un indirizzo di memoria valido, o almeno un indirizzo nel quale il programma può accedere, ragion per cui il programma crasha. Ma, oltre ad una semplice sequenza di "A", possiamo anche inserire un buffer costruito apposta, che inietta nel registro un indirizzo valido che punta ad un codice arbitrario. Siamo quindi nella situazione di un *buffer overflow* sfruttato in modo da poter eseguire codice arbitrario sul sistema, codice che può mirare ad aggiungere un nuovo utente con certi privilegi su quel sistema, ad ottenere i privilegi di amministratore in modo indebito o ad aprire una shell remota o locale in modo indebito. In ogni caso, quando un attaccante ha sfruttato un codice vulnerabile iniettando del codice arbitrario al suo interno ha il controllo totale della macchina, anche se indebito. I bollettini di sicurezza in giro per il web pullulano di bug del genere trovati ancora oggi in molte applicazioni, e dovuti proprio all'uso errato di funzioni come quelle che abbiamo visto sopra, bug che in genere sono corretti il più in fretta possibile dopo la scoperta per evitare che i danni ai sistemi che usano quelle applicazioni diventino maggiori. Non vedremo in questa sede, per evitare di divagare troppo nel discorso, in che modo sfruttare tali vulnerabilità per acquisire il controllo di un sistema, ma per ora ci basta sapere che usando certe funzioni la cosa è possibile e sapere in che modo funziona.

In conclusione, le funzioni potenzialmente vulnerabili a buffer overflow e da usare

con cautela sono:

- scanf
- gets
- strcpy
- strcat
- sprintf

Queste funzioni vanno usate solo quando si è sicuri al 100% delle dimensioni del buffer di destinazione. In alternativa, è più sicuro usare funzioni come

- fgets
- strncpy
- strncat
- snprintf

Soffermiamoci un attimo sulla fgets (ne faremo solo una trattazione sommaria per le stringhe in questa sede, mentre la studieremo in modo più approfondito nel capitolo sui file). Abbiamo visto prima che, per la lettura di una stringa da input, sia l'uso di scanf che di gets è pericoloso. Per leggere stringhe la cosa migliore è fare ricorso a questa funzione, che prende come primo argomento la stringa di destinazione, come secondo argomento il numero massimo di caratteri da leggere da input e come terzo argomento il descrittore da cui leggere (nel nostro caso lo standard input, identificato da stdin). Esempio di uso:

```
char str[16];

printf ("Inserisci una stringa: ");
fgets (str,sizeof(str),stdin);
str[strlen(str)-1]=0;

printf ("Stringa inserita: %s\n",str);
```

La notazione sizeof(str) dice di leggere da input al massimo tanti caratteri quanti sono quelli supportati dalla dimensione di str (ovvero 16 in questo caso), mentre stdin, costante definita in stdio.h, identifica lo standard input. La scrittura `str[strlen(str)-1]=0;` serve perché la funzione fgets salva nella stringa anche la pressione del carattere invio. Questa scrittura setta il carattere NULL un byte prima, in modo da rimuovere il carattere invio dalla stringa.

Attenzione anche ad evitare scritture del genere:

```
char *str = "Ciao";
printf (str);
```

Se non viene specificato esplicitamente il formato della stringa da stampare nella printf l'applicazione può potenzialmente essere vulnerabile a *format string overflow*, una vulnerabilità scoperta abbastanza recentemente che consente di scrivere dati arbitrari sullo stack. Seguendo questi passi per evitare buffer overflow e format string overflow si può essere sicuri almeno a un 70% di scrivere applicazioni relativamente

sicure.

Algoritmi di ordinamento

.

Una delle caratteristiche irrinunciabili in un calcolatore è la capacità di *ordinare* dati. È così irrinunciabile che il nome che i francesi danno al computer moderno è *ordinateur*, ordinatore. Gli informatici nel corso degli anni hanno studiato e messo a punto molti algoritmi di ordinamento, ovvero algoritmi in grado di ordinare insiemi di dati (nel nostro caso array). Ciò che differenzia un algoritmo dall'altro è il suo grado di ottimizzazione, ovvero il numero medio di passi compiuti per giungere allo scopo finale (ovvero avere un vettore ordinato in senso crescente o decrescente), e spesso e volentieri un algoritmo abbastanza immediato per il nostro modo di ragionare non lo è per il calcolatore, e viceversa. Ecco che la necessità di risparmiare in fatto di tempo di esecuzione del codice sul calcolatore (necessità che diventa irrinunciabile quando si deve ordinare una grande mole di dati) ha portato col tempo allo sviluppo di algoritmi di ordinamento via via più complessi per la logica umana, ma estremamente ottimizzati per il calcolatore. In questa sede prenderemo in esame gli algoritmi più usati, andando in ordine crescente in quanto a complessità (e decrescente in quanto a ottimizzazione):

Naive sort

Si tratta dell'algoritmo di ordinamento più semplice e anche meno ottimizzato per il calcolatore. Quello che fa è trovare in un vettore la posizione dell'elemento più grande. Se la sua posizione non è alla fine del vettore (infatti in un vettore ordinato in modo crescente l'elemento più grande si trova alla fine) allora scambia tra di loro l'elemento all'ultima posizione e il valore massimo, in modo che l'elemento più grande si trovi all'ultima posizione. All'iterazione successiva viene considerato il vettore come di dimensione $dim-1$, dove dim è la dimensione di partenza. Vengono effettuate tali iterazioni finché la dimensione del vettore non è uguale a 1 (ovvero il vettore è ordinato). Esempio pratico dell'algoritmo:

$v = \{1,0,5,4\}$

$v = \{1,0,4,5\}$

$v = \{0,1,4,5\}$

Ed ecco come scriverlo in C (esempio applicato a un vettore di interi):

```
// Procedura per lo scambio dei valori tra due variabili
```

```

void swap (int *a, int *b) {
    int *tmp;

    tmp=a;
    a=b;
    b=tmp;
}

int findPosMax(int *v, int n) {
    int i,p=0; /* ipotesi: max = v[0] */

    // Ciclo su tutti gli elementi dell'array
    for (i=1; i<n; i++)
        // Se l'elemento attuale è maggiore dell'elemento massimo,
        // allora il nuovo indice del massimo è quello appena trovato
        if (v[p]<v[i]) p=i;
    return p;
}

void naiveSort(int *v, int dim) {
    int p;

    // Finché nel vettore ci sono elementi...
    while (dim>1) {
        // ...trova la posizione dell'elemento più grande
        p = findPosMax(v, dim);

        // Se la sua posizione non è alla fine del vettore,
        // scambia tra di loro l'elemento massimo e l'ultimo elemento
        if (p < dim-1) scambia(&v[p],&v[dim-1]);

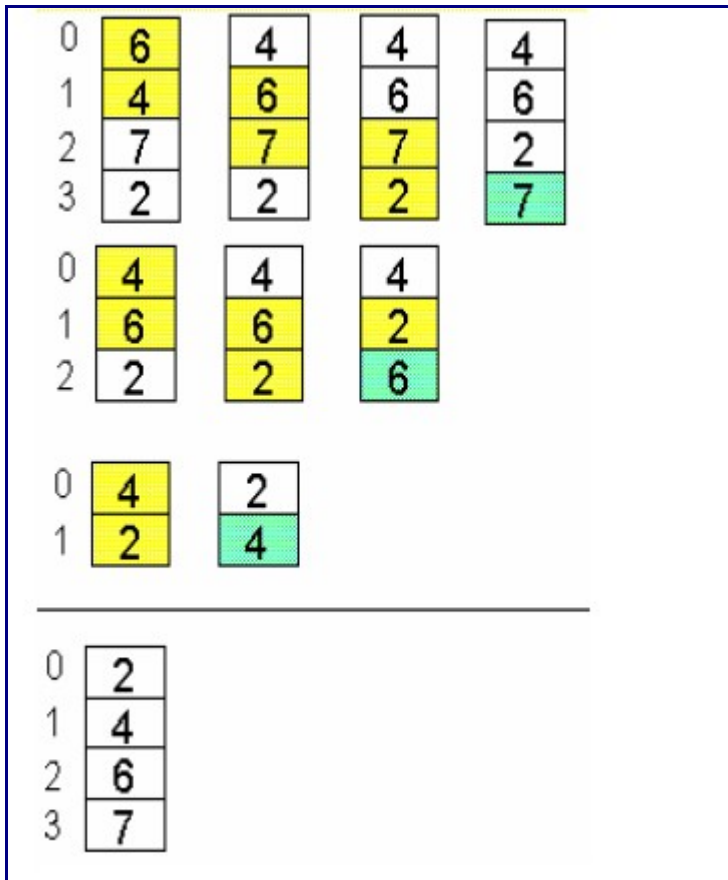
        // Decrementa la dimensione del vettore
        dim--;
    }
}

```

Bubble sort

Il bubble sort è un algoritmo più efficiente del naive anche se leggermente meno intuitivo. Il difetto principale del naive sort è infatti quello che non si accorge quando il vettore è già ordinato, e in tal caso continua a effettuare iterazioni su di esso. Il bubble sort corregge questo difetto considerando coppie adiacenti di elementi nel vettore, e non il vettore nella sua interezza, e partendo dal presupposto che il vettore sia ordinato. Se due coppie adiacenti qualsiasi sono scambiate tra di loro (prima il valore più grande e poi quello più piccolo) effettua uno scambio, e quindi vuol dire che il vettore non era ordinato. Se invece non si verifica alcuno scambio il vettore è già ordinato, e quindi l'algoritmo termina.

Esempio applicativo:



Ecco un codice dell' algoritmo:

```
// Prende come argomenti il vettore da ordinare e la sua dimensione
void bubbleSort(int *v, int dim){
    int i;
    bool ordinato = false;

    // Finché ci sono elementi nel vettore e il vettore non è
    ordinato...
    while (dim>1 && !ordinato) {
        // Ipotesi: vettore ordinato
        ordinato = true;

        // Per tutti gli elementi nel vettore
        for (i=0; i<dim-1; i++)
            // Se l'i-esimo elemento è maggiore dell'i+1-esimo elemento...
            if (v[i]>v[i+1]) {
                // ...scambia tra di loro i due elementi
                swap(&v[i],&v[i+1]);

                // Il vettore NON è ordinato
                ordinato = false;
            }
    }
}
```

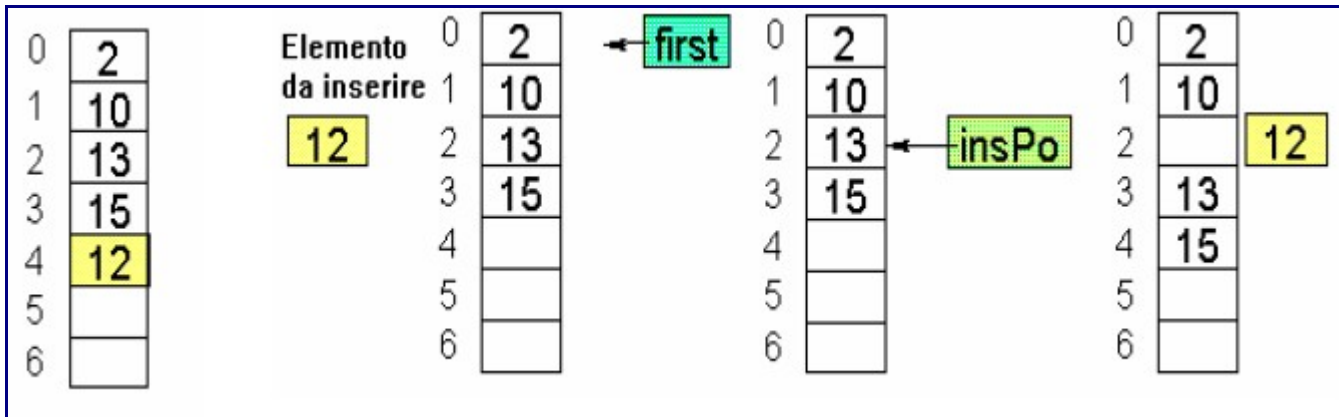
```

    // Considera il vettore come di dimensione dim-1
    dim--;
}
}

```

Insert sort

L'insert sort è un algoritmo che parte da un approccio diverso da quelli visti finora: per ottenere un vettore ordinato basta *costruirlo* ordinato, inserendo ogni elemento al posto giusto. Ecco un esempio grafico:



Per implementarlo useremo due funzioni. La funzione *insertSort* prende come parametri il vettore da ordinare e la sua dimensione, e, per i che va da 0 a $N-1$, inserisce alla posizione corretta all'interno del sottovettore $v[0], \dots, v[i]$ l' i -esimo elemento del vettore:

```

void insertSort(int *v, int dim) {
    int i;

    // Ciclo su tutti gli elementi
    for (i=1; i<dim; i++)
        // Inserisco al posto giusto l'i-esimo elemento
        insMinore(v,i);
}

```

La funzione *insMinore* prende come parametri il vettore e la posizione dell'elemento da ordinare. Questa funzione determina la posizione in cui va inserito l'elemento alla posizione specificata, crea lo spazio per l'inserimento spostando gli elementi all'interno del vettore ed effettua l'inserimento:

```

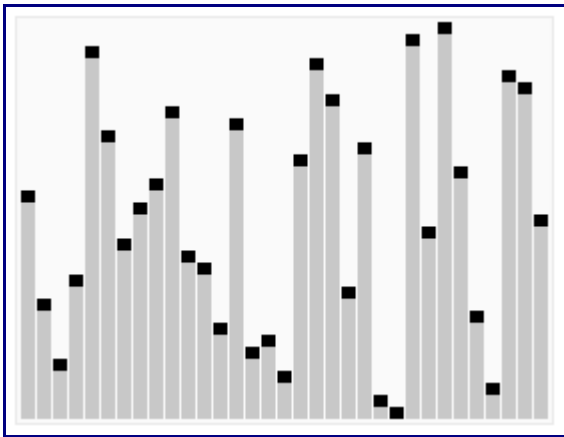
void insMinore(int *v, int lastpos) {
    int i, x = v[lastpos];

    for (i = lastpos-1; i>=0 && x<v[i]; i--)
        v[i+1]= v[i]; /* crea lo spazio */
    v[i+1]=x;
}

```

Quick sort

Avvicinandoci via via ad algoritmi sempre più ottimizzati giungiamo al *quick sort*, algoritmo di default per l'ordinamento usato ancora oggi dal C. Il quick sort si basa su un principio relativamente semplice: ordinare un vettore di piccole dimensioni è molto meno costoso dell'ordinare un vettore di grandi dimensioni. L'idea è quella di dividere il vettore di principio in due sottovettori, con un elemento intermedio (chiamato *pivot*). Le celle di memoria prima del pivot conterranno tutti gli elementi minori del pivot, quelle successive gli elementi maggiori del pivot. A questo punto l'algoritmo viene applicato ricorsivamente ai due sottovettori, fino ad arrivare a vettori di dimensione unitaria che, per definizione, sono già ordinati. Ecco una piccola animazione che illustra il funzionamento:



Ed ecco una possibile specifica:

```
void qSort(int v[], int first, int last){
    if (vettore non vuoto)
        <scegli come pivot l'elemento medio>
        <isola nella prima metà vettore gli
        elementi minori o uguali al pivot e
        nella seconda metà quelli maggiori>
        <richiama quicksort ricorsivamente
        sui due sottovettori >
}
```

Codice:

```
void qSort(int *v, int first, int last){
    int i,j,pivot;

    if (first<last) {
        // Partenza: i parte dal primo elemento del vettore, j
dall'ultimo
        i = first; j = last;

        // Il pivot è l'elemento medio del vettore
        pivot = v[(first + last)/2];
```

```

do {
    // Finché l'elemento generico i-esimo a sinistra del pivot
    // è minore del pivot, incrementa i
    while (v[i] < pivot) i++;

    // Finché l'elemento generico j-esimo a destra del pivot
    // è maggiore del pivot, decrementa j
    while (v[j] > pivot) j--;

    // Altrimenti, scambia tra loro l'elemento i-esimo e quello j-
esimo
    if (i <= j) {
        swap(&v[i], &v[j]);
        i++, j--;
    }
} while (i <= j); // Cicla finché i e j non si incontrano

// Richiama il quick sort sul primo sottovettore
qSort(v, first, j);

// Richiama il quick sort sul secondo sottovettore
quickSort(v, i, last);
}
}

```

Tipi di dato derivati, enumerazioni e strutture

.

I tipi di dato su cui abbiamo operato finora erano tipi di dati *semplici*. Abbiamo infatti operato su variabili sia scalari sia vettoriali, ma tutte identificate univocamente da un tipo (una variabile int, una variabile float, un array di int, un array di char...). Il C, al pari degli altri linguaggi di programmazione ad alto livello, consente anche al programmatore di definire dei propri tipi di dato e di operare su tipi di dato *composti* (appunto le strutture, chiamate record nell'informatica teorica), ovvero tipi di dato composti da tipi di variabili eterogenei.

Definire propri tipi - L'operatore typedef

Accennavamo prima alla possibilità di poter definire propri tipi di dato in C, a seconda delle esigenze del programmatore. Il C mette a disposizione l'operatore *typedef* per definire nuovi tipi a partire dai tipi primitivi già esistenti.

Sia ben inteso, non è indispensabile la definizione di nuovi tipi di dato in un programma. Si possono benissimo manipolare dati primitivi anche in un programma di grandi dimensioni, o usare strutture specificando in modo esplicito l'etichetta *struct*, come vedremo in seguito, ma l'uso di *typedef* rende la scrittura del programma più intuitiva (se un tipo di variabile la uso solo per la temperatura posso chiamare il suo tipo *temp*, il che rende il suo uso più intuitivo rispetto a un semplice float) e probabilmente più leggibile.

Esempio di utilizzo dell'operatore *typedef*: voglio creare un nuovo tipo di variabile solo per misurare gli angoli in gradi, a partire dal tipo float. Ricorrerò ad una scrittura del tipo

```
typedef float degree;
```

Ora posso sfruttare il nuovo tipo nel programma:

```
degree alpha=90;
```

```
printf ("L'angolo alpha è di %f gradi\n",alpha);
```

Un altro utilizzo molto comodo è per la definizione di nuovi dati di tipo vettoriale. Ad esempio, so che un codice fiscale è sempre composto da 17 caratteri. Posso creare un nuovo tipo di dato dedicato alla memorizzazione dei codici fiscali in questo modo:


```
typedef char CF[17];  
  
.....  
  
CF c = "AAABBBCCDDDEEEFF";
```

Le dichiarazioni dei nuovi tipi in genere vanno messe in modo da essere visibili a tutte le funzioni del programma, quindi o al di fuori del main o in un file header importato dall'applicazione.

Enumerazioni

Le enumerazioni in C si dichiarano attraverso la keyword *enum*, e hanno l'obiettivo di dichiarare nuovi tipi di dato con un dominio limitato, dove al primo valore dell'enumerazione viene associato il valore 0, al secondo il valore 1 e così via.

Ad esempio, in C non ho di default un tipo di dato per poter operare su tipi booleani. Posso però costruirmi un tipo di dato booleano grazie ad un enumerazione:

```
typedef enum { false, true } boolean;
```

In questo caso il primo campo dell'enumerazione è *false*, a cui viene attribuito il valore 0, e il secondo è *true*, a cui quindi viene attribuito il valore 1. Ora, grazie alla specifica typedef, posso usare questo tipo di dato all'interno del mio codice:

```
boolean trovato=false;  
  
.....  
  
if (valore1==valore2)  
    trovato=true;
```

Altro esempio di enumerazione:

```
typedef enum {  
    Lunedì,  
    Martedì,  
    Mercoledì,  
    Giovedì,  
    Venerdì,  
    Sabato,  
    Domenica  
} giorni;
```

In questo caso Lunedì=0, Martedì=1, ..., Domenica=6. All'interno del mio codice posso istanziare una variabile di questo tipo e sfruttarla così:

```
giorno g1=Lunedì;  
giorno g2=Martedì;  
.....
```

Dati strutturati

Nella realtà di tutti i giorni abbiamo a che fare con entità descritte da più di una caratteristica, e anche con tipi diversi di caratteristiche. Per soddisfare questa esigenza, il C mette a disposizione i *tipi strutturati*. Esempio classico di tipo strutturato: un'automobile è descritta da una targa, dall'anno di immatricolazione, dalla casa produttrice e dal modello. Ecco come implementare queste caratteristiche in C, creando il tipo di dato strutturato 'automobile':

```
typedef struct {
    char targa[16];
    char marca[16];
    char modello[16];
    int anno_imm;
} automobile;
```

Usando la keyword typedef, posso usare questo tipo di dato all'interno del mio programma direttamente così:

```
automobile a;
```

In alternativa, potevo specificare la struttura senza typedef:

```
struct automobile {
    char targa[16];
    char marca[16];
    char modello[16];
    int anno_imm;
};
// Ricordate sempre il ; finale
```

In questo caso, posso usare il tipo di dato strutturato all'interno del mio programma ma specificando anche il fatto che faccio uso di un tipo di dato strutturato dichiarato in precedenza:

```
struct automobile a;
```

Per comodità e maggiore leggibilità del codice, in questo luogo useremo la prima scrittura (quella con il typedef).

Per accedere ai dati contenuti all'interno di una struttura posso sfruttare un'istanza della struttura stessa (ad esempio, nel caso di sopra, una variabile di tipo 'automobile') e specificare il componente a cui voglio accedere separato da un punto '.'. Esempio:

```
typedef struct {
    char targa[16];
    char marca[16];
    char modello[16];
    int anno_imm;
} automobile;
```

```

.....

automobile a;

printf ("Inserisci la targa: ");
scanf ("%s",a.targa);

printf ("Inserisci la marca: ");
scanf ("%s",a.marca);

printf ("Inserisci il modello: ");
scanf ("%s",a.modello);

printf ("Inserisci l'anno di immatricolazione: ");
scanf ("%d",&a.anno_imm);

printf ("Targa: %s\n",a.targa);
printf ("Marca: %s\n",a.marca);
printf ("Modello: %s\n",a.modello);
printf ("Anno di immatricolazione: %d\n",a.anno_imm);

```

È possibile anche definire array di tipi strutturati, in questo modo:

```

// Array di 10 automobili
automobile a[10];
int i;

for (i=0; i<10; i++) {
    printf ("Automobile n.%d\n\n",i+1);

    printf ("Inserisci la targa: ");
    scanf ("%s",a.targa);

    printf ("Inserisci la marca: ");
    scanf ("%s",a.marca);

    printf ("Inserisci il modello: ");
    scanf ("%s",a.modello);

    printf ("Inserisci l'anno di immatricolazione: ");
    scanf ("%d",&a.anno_imm);
}

```

e ovviamente vale lo stesso discorso fatto con gli array di tipi primitivi per quanto riguarda l'inizializzazione dinamica:

```

// Puntatore alla struttura automobile
automobile *a;
int i,n;

printf ("Inserire i dati di quante automobili? ");
scanf ("%d",&n);

```

```

// Inizializzazione dinamica del vettore di automobili
a = (automobile*) malloc (n*sizeof(automobile));

for (i=0; i<n; i++) {
    printf ("Automobile n.-%d\n\n",i+1);

    printf ("Inserisci la targa: ");
    scanf ("%s",a.targa);

    printf ("Inserisci la marca: ");
    scanf ("%s",a.marca);

    printf ("Inserisci il modello: ");
    scanf ("%s",a.modello);

    printf ("Inserisci l'anno di immatricolazione: ");
    scanf ("%d",&a.anno_imm);
}

```

Posso anche dichiarare puntatori a strutture e accedere alle strutture stesse tramite questi puntatori. In questo caso, invece del punto '.' per accedere ad un certo elemento della struttura il C propone un operatore apposito, l'operatore '->':

```

// Puntatore a struttura
automobile *a;

printf ("Inserisci la targa: ");
scanf ("%s",a->targa);

printf ("Inserisci la marca: ");
scanf ("%s",a->marca);

printf ("Inserisci il modello: ");
scanf ("%s",a->modello);

printf ("Inserisci l'anno di immatricolazione: ");
scanf ("%d",&a->anno_imm);

printf ("Targa: %s\n",a->targa);
printf ("Marca: %s\n",a->marca);
printf ("Modello: %s\n",a->modello);
printf ("Anno di immatricolazione: %d\n",a->anno_imm);

```

Direttive per il preprocessore

.

Ogni compilatore traduce le istruzioni di un file sorgente in linguaggio macchina. Il programmatore generalmente non è consapevole del lavoro del compilatore: si fornisce delle istruzioni di un linguaggio di alto livello per evitare le complessità gestionali del linguaggio macchina. Ma, comunque, è importante poter comunicare con il compilatore. Il C fa uso del preprocessore per estendere la sua potenza e la sua notazione, consentendo al programmatore un'interazione con il compilatore. L'identificatore delle righe che riguardano le direttive ad esso è #, che nel C ANSI può essere anche preceduto da spazi mentre nel C Tradizionale deve trovarsi all'inizio della riga. Le direttive non fanno comunque parte della grammatica del linguaggio, ampliano solo l'ambiente di programmazione. Per lo standard ANSI, le direttive sono le seguenti:

```
#define #error #include #elif #if
#line #else #ifdef #pragma #endif
#ifdef #undef #warning
```

La direttiva #include

Solitamente anche nei programmi più banali si usa la direttiva **#include** per, appunto, includere nel sorgente file esterni o librerie.

Per includere una libreria si usano le parentesi angolari < e >, mentre per includere un file esterno o magari nella stessa cartella del programma si usano i doppi apici " .

Un esempio di inclusione di una libreria e un file che si trova nella cartella superiore di dove si trova il sorgente in cui la includiamo:

```
#include <stdio.h>
#include "../file1.h"
```

In questo caso il preprocessore quando incontrerà queste righe le sostituirà con il contenuto del file richiamato.

In [Unix](#) solitamente i file d'intestazione specificati nelle parentesi angolari si trovano nel percorso **/usr/include/**.

Nei file inclusi possono naturalmente anche esserci altre direttive al preprocessore che verranno poi a loro volta "lavorate".

La direttiva #define

La direttiva **#define** si usa, appunto, per definire qualcosa ad esempio:

```
#define scrivi printf
```

In questo caso la definizione è **scrivi** che va a sostituire la parola **printf** quindi nel corso del programma al posto di:

```
printf("Ciao preprocessore!");
```

Si potrà scrivere

```
scrivi("Ciao preprocessore!");
```

Comunque il define può anche definire numeri, simboli o altro. Vari esempi di define:

```
#define EQ ==  
#define OK printf("OK\n");  
#define DEBUG 1
```

Nell'esempio sopra visualizzato si può notare il cosiddetto "ZUCCHERO SINTATTICO" dato che ogni tanto a un programmatore in C può scappare di mettere un solo = nelle uguaglianze così con la definizione *EQ ==* si potrà scrivere così:

```
if ( a EQ b ) ...
```

Evitando errori di sintassi.

Un'altra cosa da notare è la definizione *DEBUG* molto utile nelle fasi di test di un programma che si può usare nel controllo del flusso tramite sempre direttive al preprocessore che vedremo adesso.

Controllo del flusso

Con le direttive al preprocessore si può eseguire anche un flusso del controllo (*if, else*) utilizzando le direttive **#if**, **#else**, **#elif**, **#endif** e **#ifdef**.

Iniziamo a spiegarli dai primi cioè **#if**, **#else**, **#elif** e **#endif** che corrispondono al controllo del flusso normalmente utilizzato: **if**, **else**, **else if** mentre l'ultimo **#endif** è "originale" del preprocessore.

Esempio:

```
#include <stdio.h>  
#define A 2  
#define B 4  
  
int main(void)
```

```

{
    #if A == 2 || B == 2
        printf("A o B sono uguali a 2\n");
    #elif A == 4 && B == 4
        printf("A e B sono uguali a 4\n");
    #elif A != B
        printf("A è diversa da B\n");
    #else
        printf("A e B sono di un valore non definito\n");
    #endif

    return 0;
}

```

Si possono notare le seguenti cose:

- Le variabili su cui eseguire controlli devono essere definite tramite **#define**
- Anche nel controllo del flusso tramite direttive al preprocessore si possono eseguire controlli con **||** (*OR*), **&&** (*AND*) e **!=** (*NOT*).
- La direttiva **#endif** "dice" al preprocessore che il controllo del flusso è finito.

Per eseguire un debug con questo sistema si potrebbe inserire qualcosa tipo:

```

#if DEBUG 1
    printf("x = %d\n", x);
    printf("y = %s\n", y);
    ...
#endif

```

Ma ora vedremo con la direttiva **#ifdef** cosa si può fare, in pratica "**ifdef**" sta per "**se definito**" quindi si può tramite essa controllare se una variabile è stata definita o meno e con l'aggiunta delle direttive **#undef** e **#ifndef** vedremo cosa si può fare con l'esempio seguente:

```

#include <stdio.h>
#define NUMERO 4

int main(void)
{
    #ifndef NUMERO
        #define NUMBER 4
    #ifdef NUMBER
        #undef NUMBER
        #define NUMERO 4
    #endif
    return 0;
}

```

Innanzitutto chiariamo cosa vuol dire *ifndef* e *undef*, la prima equivale a "**se non è definito**" (*if not define*) mentre la seconda equivale a "**togli la definizione**" (*undefine*).

Nell'esempio sopra definiamo **NUMERO** dopodichè all'interno del corpo main iniziamo col verificare se non è definito numero, se ciò è vero definiamo NUMBER, se invece è definito NUMBER togliamo la definizione di NUMBER e definiamo NUMERO. Dopodichè si esce dal programma.

La direttiva **#undef** diciamo che è inutile nei piccoli programmi, ma risulta utilissima nei programmi di grandi dimensioni composti magari da molti file e da molte persone che ci lavorano e senza andare in giro o sfogliare tra i file se una cosa è stata definita o meno questa semplice direttiva ci facilita la vita.

L'uso di **#ifdef** è utilissimo nel caso in cui si vogliono usare dei file header. Infatti, un file header potrebbe essere incluso in due diversi file sorgenti che si vanno a compilare insieme, e questo potrebbe generare ambiguità ed errori in fase di compilazione (funzioni o dati che risulterebbero dichiarati due volte). Per evitare questo problema si usano proprio le direttive al preprocessore. Nell'header che andremo a creare avremo una cosa del genere:

- Se la variabile `_NOMEHEADER_H` non è definita
- Definisci la variabile
- Dichiarati tutto il contenuto dell'header
- Altrimenti, termina la dichiarazione dell'header

In codice:

```
#ifndef _MIOHEADER_H
#define _MIOHEADER_H

// Qui metto tutte le mie funzioni e i miei dati

#endif
```

In pratica, una volta definita la macro `_MIOHEADER_H` il file header non verrà più incluso in nessun altro file, risolvendo quindi gli eventuali problemi di header definiti due o più volte.

Macro con parametri

Con la direttiva **#define** si possono definire macro con parametri. Il "come" lo vedremo nell'esempio seguente:

```
#define PD(x) ((x) * (x))
```

PD sta per "**P**otenza di **D**ue" in pratica se nel corso del programma si esegue qualcosa tipo:

```
a = PD(3);
```

il preprocessore la espanderà in questo modo:

```
a = ((3) * (3));
```


Un'altro esempio per concludere aggiungerei:

```
#define minore(x, y) ((x) < (y)) ? (x) : (y)
```

Questa definizione una volta richiamata rilascia il valore più piccolo tra i due numeri dati. Naturalmente l'utilizzo espanso di parentesi consente di essere sicuri che una volta espansa la stringa tutte le operazioni vengano eseguite nel modo desiderato.

Macro predefinite

Nel C esistono 5 tipi di macro già definite sempre disponibili che non possono essere ridefinite dal programmatore. Si possono vedere nello schema seguente:

```
/* MACRO    ||  COSA CONTIENE */
__DATE__   /* Una stringa che contiene la data corrente */
__FILE__   /* Una stringa che contiene il nome del file */
__TIME__   /* Una stringa che contiene l'ora corrente */
__LINE__   /* Un intero che rappresenta il numero di riga
corrente */
__STDC__   /* Un intero diverso da 0 se l'implementazione segue
lo standard ANSI C */
```

Operatori # e

Questo tipo di operatori sono disponibili solo nel C ANSI. L'operatore unario # trasforma un parametro formale di una definizione di macro in una stringa ad esempio:

```
#define nomi(a, b) printf("Ciao " #a " e " #b "! Benvenuti!\n");
```

da richiamare nel corpo main con:

```
nomi(HdS619, BlackLight);
```

Una volta espanso dal preprocessore questa linea diventerà:

```
printf("Ciao " "HdS619" " e " "BlackLight" "! Benvenuti!\n");
```

Ora invece vediamo l'operatore binario ## che serve a concatenare token. Ad esempio:

```
#include <stdio.h>
#define X(y) x ## y
X(3) = X(4) = X(12) = ...
```

verrà espanso in:

```
x3 = x4 = x12 = ...
```

In pratica si può pensare che "colleghi" i due parametri **x** e **y**.

Direttive **#error** e **#warning**

Le direttive **#error** e **#warning** servono rispettivamente per dare errori nella compilazione oppure avvisi. Solitamente queste due direttive vengono usate insieme a quelle che controllano il "flusso di compilazione" (**#else**, **#if**, **#undef**, ecc...). La loro sintassi è la seguente:

```
#error   Messaggio di errore  
#warning Messaggio di avvertimento
```

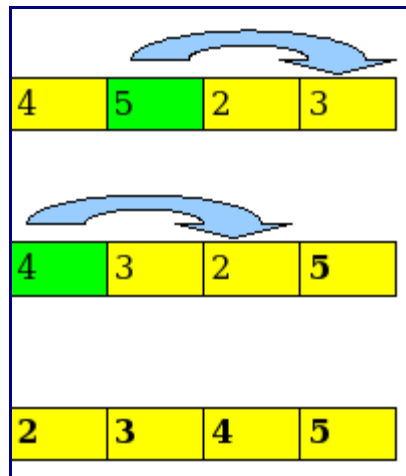
Funzione ricorsive

.

A volte capita di avere a che fare con problemi che sono difficilmente risolvibili ricorrendo a funzioni imperative “standard”. In alcuni casi, invece di avere una visione “di insieme” del problema da risolvere può essere più comodo avere una visione “particolareggiata”, progettare un algoritmo che risolva parte del problema e ripetere quest'algoritmo finché il problema non è risolto del tutto.

Esempio informale di ricorsione

Un esempio pratico: immaginiamo di dover ordinare un array di numeri in senso crescente. La soluzione che ci viene in mente ora, senza applicare algoritmi ricorsivi, è quella di cercare il valore più grande all'interno dell'array, spostarlo nell'ultima posizione e poi ordinare l'array escludendo il termine appena “ordinato”, e ripetere questa procedura finché l'array non contiene più nessun elemento da ordinare.



Questo modo però implica una visione “di insieme” del problema, e per questo non è la più efficiente (è un algoritmo chiamato naive sort).

E se invece dividessimo via via l'array in parti più piccole, fino ad arrivare ad array contenenti ognuno due elementi? Potremmo ordinare ognuno di questi mini-array (si tratterebbe al massimo di fare uno scambio tra due elementi), quindi ricorsivamente in questo modo risalire ad un array ordinato. Questa è la soluzione più ottimizzata in termini di prestazioni, ed implica un nuovo approccio alla risoluzione di un problema: un approccio ricorsivo. Dal particolare (l'ordinamento di array di due elementi) si passa al generale (l'ordinamento di un intero array di dimensioni

maggiori), facendo in modo che la funzione di ordinamento richiami sempre se stessa (questo è un algoritmo di merge sort, implementato di default in linguaggi come Java e Perl).

Esempio pratico di ricorsione

Facciamo un esempio pratico di ricorsione: il classico calcolo del fattoriale. Il fattoriale di un numero intero n è $n! = n*(n-1)*(n-2)*...*1$

Con i cicli classici che abbiamo visto finora potremmo scriverlo così:

```
/* Questo è il main() */

main() {
    int n;

    printf ("Inserire un numero intero: ");
    scanf ("%d",&n);

    printf ("Fattoriale di %d: %d\n",n,fat(n));
}

/* Questa è la funzione che calcola il fattoriale */

int fat(int n) {
    int i,f=1;

    for (i=n; i>0; i--)
        f *= i;
    return f;
}
```

Vediamo ora come riscrivere la funzione fat() in modo ricorsivo, senza nemmeno usare il ciclo for. Di volta in volta la variabile di appoggio i viene decrementata di un'unità. Proviamo invece a ragionare in modo ricorsivo:

Ho una variabile n di cui voglio calcolare il fattoriale:

$$n! = n*(n-1)*(n-2)*...*1$$

$$\text{Ma } (n-1)! = (n-1)*(n-2)*...*1 \rightarrow n! = n*(n-1)! \text{ Ma } (n-2)! = (n-2)*(n-3)*...*1 \rightarrow (n-1)! = (n-1)*(n-2)! \text{ E così via}$$

Per calcolare il fattoriale di n posso quindi semplicemente moltiplicare n per il fattoriale di $n-1$, che a sua volta è $n-1$ moltiplicato per il fattoriale di $n-2$, e così via finché non arrivo a 1.

Ecco l'implementazione:

```
int fat(int n) {
    if (n==1)
        return 1;
```

```

    else return n*fat(n-1);
}

```

Un'implementazione molto più semplice e immediata.

Ricorsione tail e non-tail

Una forma di questo tipo di definisce una forma ricorsiva di tipo non-tail. Una forma ricorsiva si definisce di tipo non-tail quando nella direttiva di ritorno (return) non compare solo la chiamata alla funzione ricorsiva, ma anche un parametro (in questo caso n , che viene moltiplicato per la funzione ricorsiva). Quando invece nella direttiva di ritorno è presente solo la chiamata alla funzione ricorsiva, allora abbiamo a che fare con una forma ricorsiva di tipo tail. Facciamo un esempio di funzione che sfrutti una ricorsione di tipo tail. Vogliamo creare una funzione che, dato un array di interi, ritorna il numero di elementi nulli al suo interno. Potremmo anche crearla in modo “standard”, con un normale ciclo for o con un ciclo while:

```

/*
La funzione countNull accetta come parametri un vettore di
interi e la dimensione del vettore stesso, e ritorna il numero
di zeri contenuti all'interno del vettore
*/

int countNull ( int *v, int dim ) {
    int i=0,count=0;

    // Se il vettore non ha elementi, ritorna 0

    if (!dim)
        return 0;
    else
        // Finché il vettore ha elementi, controllo se
        // l'elemento è zero. Se sì, incremento la variabile
        // contatore

        for (i=0; i<dim; i++)
            if (!v[i])        count++;
    return count;
}

```

Ecco invece come strutturare la funzione con una ricorsione tail:

Se la posizione attuale all'interno del vettore è l'ultima, ritorna il numero di zeri contati nel vettore. Se alla posizione attuale all'interno del vettore corrisponde uno zero, incrementa la variabile contatore. Ritorna la funzione stessa sullo stesso vettore della stessa dimensione ma sull'elemento successivo nel vettore.

```

int countNull(int *v, int dim, int i) {
    if (i==dim)
        return zero;
}

```

```
    if (v[i]==0)
        zero++;
    return countNull(v,dim,i+1);
}
```

In questo caso, quando richiamiamo la funzione dobbiamo anche specificare il valore iniziale della variabile *i*. Poiché vogliamo cominciare dall'inizio del vettore, *i* varrà 0.

Liste

.

Una lista è un insieme *finito e ordinato* di elementi di un certo tipo. In informatica una lista si indica come un insieme di termini compresi tra parentesi quadre []. Esempio, ['a','n','c']. Come tutti i tipi di dato astratti, anche le liste sono definite in termini di

- Dominio-base dei suoi elementi (interi, caratteri, stringhe...)
- Operatori di costruzione della lista
- Operatori di selezione sulla lista

Il grosso vantaggio delle liste sugli array è il fatto che una lista si può definire in modo estremamente dinamico, anche senza conoscere il numero di elementi totale di partenza dei suoi elementi, e di gestire i collegamenti tra un elemento e un altro in modo estremamente versatile. Ma andiamo con ordine.

Liste come tipi di dato astratto

Pochi linguaggi offrono di default il tipo 'lista' preimpostato (LISP, Prolog). Negli altri linguaggi, come C, è necessario costruirsi questo tipo in base alle proprie esigenze.

Le caratteristiche generali di un tipo di dato astratto sono state illustrate sopra. In modo più preciso, possiamo definire un tipo di dato astratto in termini di

- Dominio base D
- Insieme di funzioni $\mathfrak{S} = \{F_1, \dots, F_n\}$ sul dominio D
- Insieme di predicati $\Pi = \{P_1, \dots, P_m\}$ sul dominio D

Un tipo di dato astratto generico T è quindi definibile come

$$T = \{D, \mathfrak{S}, \Pi\}$$

Nel caso di una lista, possiamo definire

- $D = \{\text{qualsiasi tipo di dato}\}$
- $\mathfrak{S} = \{\text{cons, head, tail, emptylist}\}$
- $\Pi = \{\text{empty}\}$

Le funzioni base sulla lista sono così definite:

- $cons : D \times list \rightarrow list$
È il costruttore della lista, ovvero la funzione che, dato una lista di partenza e un elemento appartenente al dominio da inserire in cima alla lista, costruisce la lista specificata.
- $head : list \rightarrow D$
Funzione che ritorna la 'testa' della lista, ovvero il suo primo elemento.
- $tail : list \rightarrow list$
Funzione che ritorna la 'coda' della lista, ovvero una lista uguale a quella di partenza ma privata del primo elemento.
- $emptylist : \rightarrow list$
Funzione che ritorna la costante 'lista vuota'. Per convenzione, in C una lista è vuota quando il valore della sua testa è NULL.

L'unico predicato elementare sul tipo astratto di lista è così definito:

- $empty : list \rightarrow boolean$
Funzione che verifica se la lista è vuota o meno.

Qualche esempio:

- $cons(5, [3,6,2,3])$ crea la lista $[5,3,6,2,3]$
- $head([7,3,5,6])$ ritorna 7 (testa della lista)
- $tail([7,3,5,6])$ ritorna la lista $[3,5,6]$ (coda della lista)
- $empty([7,3,5,6])$ ritorna *false* (la lista non è vuota)

Quelle illustrate sono le operazioni di base che si possono effettuare su una lista. Tutte le altre operazioni (inserimento ordinato di elementi, ribaltamento degli elementi, stampa degli elementi presenti...) sono operazioni derivate dalle primitive appena illustrate. Considerando che esiste il concetto di lista vuota (per convenzione la lista avente NULL in testa) e che è possibile costruire nuove liste usando il costruttore *cons*, si possono definire tutte le eventuali funzioni derivate sulla base di quelle già definite tramite *algoritmi ricorsivi*.

Rappresentazione statica

La rappresentazione più ovvia del tipo astratto di lista è gestendo gli elementi della lista in un array. La lista così costruita conterrà

- Un vettore di lunghezza massima prefissata
- Una variabile *primo*, che identifica l'indice del primo elemento della lista
- Una variabile *lunghezza*, che indica il numero di elementi contenuti nella lista

L'inconveniente principale è il fatto che le dimensioni del vettore sono fisse. Il tipo di dato lista è quindi strutturato così in questo caso:

```
#define N 100

typedef struct {
    int primo, lunghezza;
    int elementi[N];
} list;
```

E le primitive che agiscono sulla lista sono così definite:

```
// Ritorna una lista vuota
list emptylist() {
    list l;

    // Convenzione: quando la lista è vuota l'indice del primo
    // elemento
    // è un numero negativo
    l.primo=-1;
    l.lunghezza=0;
}

// Controlla se la lista è vuota
bool empty(list l) {
    return (l.primo==-1);
}

// Ritorna il primo elemento della lista
int head (list l) {
    if (empty(l)) abort();
    return l.elementi[l.primo];
}

// Ritorna la coda della lista
list tail(list l) {
    list t=l;

    // Se la lista è vuota, esce
    if (empty(l)) abort();

    // Altrimenti, la lista t avrà come primo elemento
    // il primo di l incrementato di 1, e la lunghezza
    // di l decrementata di 1 (ovvero scarto la testa della lista)
    t.primo++;
    t.lunghezza--;
    return t;
}
```

```

// Crea una nuova lista, prendendo come parametri
// l'elemento da inserire in testa e una lista di partenza
// (eventualmente vuota)
list cons (int e, list l) {
    list t;
    int i;

    // Inserisco e in testa alla lista
    t.primo=0;
    t.elementi[t.primo]=e;
    t.lunghezza=1;

    // Copio il vettore contenuto in l nella nuova lista
    for (i=1; i<=l.lunghezza; i++) {
        t.elementi[i]=t.elementi[i-1];
        t.lunghezza++;
    }
}

```

Queste sono le funzioni primitive sulla lista. Grazie a queste è possibile costruire ricorsivamente eventuali funzioni derivate. Esempio, una funzione che stampi tutti gli elementi della lista:

```

void showList(list l) {
    // Condizione di stop: se la lista è vuota, ritorna
    if (empty(l))
        return;

    // Stampa il primo elemento della lista
    printf ("%d\n",head(l));

    // Richiama la funzione sulla coda di l
    showList(tail(l));
}

```

Rappresentazione dinamica

Una rappresentazione di liste estremamente utile è quella dinamica. In questo tipo di rappresentazione si perde ogni riferimento statico (vettori, buffer di dimensione fissa). Ogni elemento della lista contiene il suo valore e un riferimento all'elemento successivo nella lista stessa. Si crea quindi così una lista *grafica*, con **nod**i (elementi della lista) e **archi** (collegamenti tra gli elementi).

Un generico elemento della lista sarà quindi così costruito:

```

// Creo una lista di interi
// Nel caso volessi riutilizzare il codice per una lista
// di un altro tipo, mi basterà modificare il tipo element
typedef element int;

typedef struct list_element {

```

```

    element value;
    struct list_element *next;
} node;

```

Il tipo `element` mi consente di scrivere del codice estremamente modulare, in quanto semplicemente modificando il tipo potrò usare la stessa lista per memorizzare interi, float, caratteri e quant'altro. Come è possibile notare inoltre nel dichiarare la struttura `node` ho usato un'etichetta (`list_element`). Ciò è indispensabile in quanto all'interno della struttura c'è un collegamento a un elemento della struttura stessa (il prossimo elemento della lista). Ma poiché `node` non è ancora stato dichiarato a quel punto, è indispensabile mettere un'etichetta temporanea. A questo punto, con un nodo della lista così definito potrò includere al suo interno il suo stesso valore e il riferimento al prossimo elemento. Nel caso l'elemento in questione sia l'ultimo della lista, si mette come suo successore, per convenzione, il valore `NULL`.

Per una maggiore genericità del codice possiamo creare funzioni che operano sul tipo `element`, in modo che se in futuro dovessimo usare lo stesso tipo di lista creato per gestire degli interi per gestire delle stringhe basterà cambiare queste funzioni che agiscono su `element`, e lasciare inalterate le funzioni che operano sulla lista. Si comincia così a entrare nell'ottica della creazione di *codice modulare* ovvero codice che è possibile scrivere una volta e riusare più volte. Vediamo le funzioni di base che possono agire sul tipo `element` (in questo caso tipo `int`, volendo modificando il tipo basterà cambiare le funzioni):

```

bool isLess (element a, element b) { return (a<b); }
bool isEqual (element a, element b) { return (a==b); }
element get (element e) { return e; }

```

```

element readElement() {
    element e;
    scanf ("%d",&e);
    return e;
}

```

```

void printElement (element e) { printf ("%d",e); }

```

A questo punto è conveniente dichiarare il tipo lista

```

typedef node* list;

```

appunto come puntatore a un elemento di tipo `node`.

Per il tipo lista le primitive saranno le seguenti:

```

// Ritorna la costante 'lista vuota'
list emptylist() { return NULL; }

```

```

// Controlla se una lista è vuota
bool empty(list l) { return (l==NULL); }

```

```

// Ritorna la testa della lista

```

```

element head (list l) {
    if (empty(l)) abort();
    return l->value;
}

// Ritorna la coda della lista
list tail (list l) {
    if (empty(l)) abort();
    return l->next;
}

// Costruttore. Genera una lista dato un elemento
// da inserire in testa e una lista
list cons (element e, list l) {
    list t;
    t = (list) malloc(sizeof(node));

    t.value=get(e);
    t.next=l;
    return t;
}

```

Con queste primitive di base è possibile costruire qualsiasi funzione che operi sul tipo di dato 'lista'. Esempio, per la stampa degli elementi contenuti nella lista:

```

void printList(list l) {
    // Condizione di stop: lista vuota
    if (l==NULL)
        return;

    printElement(l->head);
    printf ("\n");

    // Scarto l'elemento appena stampato e
    // richiamo la funzione in modo ricorsivo
    printList(l->tail);
}

```

E allo stesso modo si possono anche definire per la ricerca di un elemento nella lista, per la lettura di un elemento all'indice i della lista e così via.

Gestione dei file ad alto livello

.

“Everything is a file!”

Questa è la frase più comune tra i sistemisti Unix quando cercano di illustrarti questo o quel dettaglio di un socket o di un dispositivo. Sui sistemi Unix ogni entità è un file, un socket di rete o locale, un processo, un dispositivo, una pipe, una directory, un file fisico vero e proprio...tutto è un file perché a livello di sistema posso scrivere e leggere su tutte queste entità con le stesse primitive (write, read, open, close). Queste sono quelle che vengono chiamate primitive a basso livello per la manipolazione dei file, a basso livello perché implementate a livello di sistema e non a livello della libreria C ad alto livello.

Ma facciamo un passo indietro. Noi siamo abituati a vedere, nella vita informatica quotidiana, un file come un'entità che contiene un certo tipo di dato. Una canzone, un'immagine, un filmato, un file di testo, la nostra tesi di laurea...tutte queste cose, in apparenza così diverse da loro, vengono trattate a livello informatico come una sola entità magica, ovvero come file.

Finora abbiamo visto come scrivere applicazioni che rimangono residenti nella memoria centrale del computer, nascono quando li eseguiamo, vengono caricati nella memoria centrale, eseguono un certo numero di operazioni e poi spariscono. Delle applicazioni del genere non sono poi molto diverse da quelle che può effettuare una semplice calcolatrice se ci pensiamo...una vecchia calcolatrice non ha memoria, non si ricorda i calcoli che abbiamo fatto e non ha traccia dei numeri che abbiamo digitato la settimana scorsa. La grande potenza dei computer, che ne ha decretato il successo già negli anni '50, è invece la capacità di poter memorizzare dati su dispositivi fissi e permanenti, non volatili come le memorie centrali, e per memorizzare questi dati c'è bisogno di ricorrere a queste entità astratte che sono i file.

Ma come fa un linguaggio di programmazione, come il C, a interagire con queste entità? Come ho già detto, una strada è quella delle primitive a basso livello, implementate a livello di kernel. Queste primitive hanno il vantaggio di essere estremamente lineari (come dicevo prima con la stessa primitiva posso scrivere su entità diverse a livello logico) e veloci. Veloci perché implementate a basso livello, marchiate a fuoco nel kernel stesso, che al momento della chiamata non le deve quindi andare a pescare da una libreria esterna. Il difetto, però, è quello della portabilità. Le funzioni write, read & co. non sono ANSI-C, perché funzionano su un kernel Unix, ma non su altri tipi di sistemi. Per rendere ANSI-C anche l'accesso ai files Kernighan e Ritchie hanno ideato delle primitive ad alto livello, indipendenti dal tipo di sistema su cui sono compilate.

Apertura dei file in C

Cominciamo a capire come un sistema operativo, e quindi anche un linguaggio di programmazione, vede un file. Un file è un'entità identificata in modo univoco da un nome e una posizione sul filesystem. Non posso interagire direttamente con l'entità presente sul filesystem, ma ho bisogno di farlo da un livello di astrazione leggermente più alto: quello dell'identificatore. Quando apro un file all'interno di una mia applicazione in C non faccio altro che associare a quel file un identificatore, che altro non è che una variabile o un puntatore di un tipo particolare che mi farà da tramite nei miei accessi al file. In ANSI-C questa variabile è di tipo FILE, un'entità definita in stdio.h, e per associarla ad un file ho bisogno di ricorrere alla funzione fopen (sempre definita in stdio.h, come tutte le funzioni che operano su entità di tipo FILE). La funzione fopen è così definita:

```
FILE* fopen(const char* filename, const char* mode);
```

dove **filename* è il nome del nostro file (può essere sia un percorso relativo che assoluto, ad es. mio_file.txt oppure /home/pippo/mio_file.txt), mentre invece **mode* mi indica il modo in cui voglio aprire il mio file. Ecco le modalità possibili:

- r Apre un file di testo per la lettura
- w Crea un file di testo per la scrittura
- a Aggiunge a un file di testo
- rb Apre un file binario per la lettura
- wb Apre un file binario per la scrittura
- ab Aggiunge a un file binario
- r+ Apre un file di testo per la lettura\scrittura
- w+ Crea un file di testo per la lettura\scrittura
- a+ Aggiunge a un file di testo per la lettura\scrittura
- r+b Apre un file binario per la lettura\scrittura
- w+b Crea un file binario per la lettura\scrittura
- a+b Aggiunge a un file binario per la lettura\scrittura

Quando non è possibile aprire un file (es. il file non esiste o non si hanno i permessi necessari per scrivere o leggere al suo interno) la funzione fopen ritorna un puntatore NULL. È sempre necessario controllare, quando si usa fopen, che il valore di ritorno non sia NULL, per evitare di compiere poi operazioni di lettura o scrittura su file non valide che rischiano di crashare il programma.

Ecco un esempio di utilizzo di fopen per l'apertura di un file in lettura:

```
#define FILE_NAME          "prova.txt"

.....

FILE *fp;

fp = fopen (FILE_NAME, "r");
```

```

if (!fp) {
    printf ("Impossibile aprire il file %s in lettura\n",FILE_NAME);
    return;
}

```

È poi buona norma eliminare il puntatore al file quando non è più necessario. Questo si fa con la funzione `fclose`, così definita:

```
int fclose(FILE *fp);
```

La funzione `fclose` ritorna 0 quando la chiusura va a buon fine, -1 negli altri casi (ad esempio, il puntatore che si prova a eliminare non è associato ad alcun file).

Scrittura su file testuali - `fprintf` e `fputs`

Vediamo ora come posso scrivere e leggere su file. In questo campo le funzioni si dividono in due tipi: quelle per scrivere e leggere su file dati binari e quelle per il testo semplice (ASCII).

Vediamo prima le funzioni ASCII. Le funzioni ASCII per scrivere e leggere su file non sono altro che specializzazioni delle corrispondenti funzioni per leggere e scrivere su `stdin/stdout`. Abbiamo quindi `fprintf`, `fscanf`, `fgets` e `fputs`.

L'uso di `fprintf` è del tutto analogo a quello di `printf`, e prende come argomenti un file descriptor (puntatore alla struttura `FILE`) e una stringa di formato con eventuali argomenti, in modo del tutto analogo a una `printf`. Esempio:

```

#define MY_FILE mio_file.txt
.....

FILE *fp;

fp = fopen (MY_FILE,"w");

if (!fp) {
    printf ("Errore: impossibile aprire il file %s in scrittura\n",MY_FILE);
    return;
}

// Scrivo su file
fprintf (fp,"Questa è una prova di scrittura sul file %s\n",MY_FILE);

```

Analogamente, si può usare anche la `fputs()` per la scrittura di una stringa su file, ricordando che la `fputs` prende sempre due argomenti (il file descriptor e la stringa da scrivere su file):

```
#define MY_FILE mio_file.txt
```

```

.....

FILE *fp;

fp = fopen (MY_FILE,"w");

if (!fp) {
    printf ("Errore: impossibile aprire il file %s in scrittura\
n",MY_FILE);
    return;
}

/* Scrivo su file */
fputs (fp,"Questa è una prova di scrittura\n");

```

Tramite la `fprintf` posso scrivere su file anche dati che poi posso andare a rileggere dopo, creando una specie di piccolo 'database di testo'. Esempio:

```

#include <stdio.h>
#include <stdlib.h>

#define USER_FILE          "user.txt"

typedef struct {
    char user[30];
    char pass[30];
    char email[50];
    int age;
} user;

int main(void) {
    FILE *fp;
    user u;

    if (!(fp=fopen(USER_FILE,"a"))) {
        printf ("Errore: impossibile aprire il file %s
in modalità append\n",USER_FILE);
        exit(1);
    }

    printf ("=> Inserimento di un nuovo utente <==\n\n");

    printf ("Username: ");
    scanf ("%s",u.user);

    printf ("Password: ");
    scanf ("%s",u.pass);

    printf ("Email: ");
    scanf ("%s",u.email);

```



```

printf ("Età: ");
scanf ("%d",&u.age);

/* Scrivo i dati su file */
fprintf (fp,"%s\t%s\t%s\t%d\n",u.user,u.pass,u.email,u.age);

printf ("Dati scritti con successo sul file!\n");
fclose (fp);
}

```

Questo produrrà un file di questo tipo:

```

username1      password1      email1  età1
username2      password2      email2  età2
.....

```

Ovvero una riga per ogni utente, dove ogni campo è separato da un carattere di tabulazione.

Lettura di file testuali - fscanf e fgets

Per leggere dati di testo semplici, come accennato prima, la libreria stdio.h mette a disposizione la funzione fscanf, la cui sintassi è molto simile a quella di scanf:

```
int fscanf (FILE *fp, char *format_string, void *arg1, ..., void *argn);
```

La funzione fscanf, esattamente come scanf, ritorna il numero di oggetti letti in caso di successo, -1 in caso di errore. Quindi possiamo struttura il nostro algoritmo in questo modo: "finché fscanf ritorna un valore > 0, scrivi i valori letti"

```

#include <stdio.h>
#include <stdlib.h>

#define USER_FILE          "user.txt"

typedef struct {
    char user[30];
    char pass[30];
    char email[50];
    int age;
} user;

int main(void) {
    FILE *fp;
    user u;
    int i=0;

    if (!(fp=fopen(USER_FILE,"r"))) {
        printf ("Errore: impossibile aprire il file %s
                in modalità read-only\n",USER_FILE);
        exit(1);
    }
}

```

```

}

while (fscanf(fp,"%s\t%s\t%s\t%d\n",
             u.user,u.pass,u.email,&u.age)>0) {
    printf ("Username: %s\n",u.user);
    printf ("Password: %s\n",u.pass);
    printf ("Email: %s\n",u.email);
    printf ("Età: %d\n\n",u.age);
    i++;
}

printf ("Utenti letti nel file: %d\n",i);
fclose (fp);
}

```

Ci sono modi alternativi per effettuare quest'operazione. Ad esempio, si potrebbero contare gli utenti semplicemente contando il numero di righe nel file, in modo del tutto indipendente dal ciclo di fscanf principale. Si tratta semplicemente di introdurre una funzione del genere:

```

...

int countLines (char *file) {
    FILE *fp;
    char ch;
    int count=0;

    if (!(fp=fopen(file,"r")))
        return -1;

    while (fscanf(fp,"%c",&ch)>0)
        if (ch=='\n')
            count++;

    return count;
}

...

i=countLines(USER_FILE);
printf ("Numero di utenti letti: %d\n",i);

```

o ancora usando, invece di ciclare controllando il valore di ritorno di fscanf, si può ciclare finché non viene raggiunta la fine del file. Per far questo si ricorre in genere alla funzione feof, funzione che controlla se si è raggiunta la fine del file puntato dal file descriptor in questione. In caso affermativo, la funzione ritorna un valore diverso da 0, altrimenti ritorna 0

```

...

int countLines (char *file) {
    FILE *fp;

```

```

char ch;
int count=0;

if (!(fp=fopen(file,"r")))
    return -1;

while (!feof(fp)) {
    if ((ch = getc(fp)) == '\n')
        count++;
}

return count;
}

...

i=countLines(USER_FILE);
printf ("Numero di utenti letti: %d\n",i);

```

Anche qui, la funzione feof si pone ad un livello di astrazione superiore a quello del sistema operativo. Infatti i sistemi operativi usano strategie differenti per identificare l'EOF (End-of-File). I sistemi Unix e derivati memorizzano a livello di filesystem la dimensione di ogni file, mentre i sistemi DOS e derivati identificano l'EOF con un carattere speciale (spesso identificato dal caratteri ASCII di codice -1). La strategia dei sistemi DOS però si rivela molto pericolosa...infatti, è possibile inserire il carattere EOF in qualsiasi punto del file, e non necessariamente alla fine, e il sistema operativo interpreterà quella come fine del file, perdendo tutti gli eventuali dati successivi.

La funzione feof si erge al di sopra di questi meccanismi di basso livello, rendendo possibile l'identificazione dell'EOF su qualsiasi sistema operativo.

Se conosco a priori la dimensione del buffer che devo andare a leggere dal file, è preferibile usare la funzione fgets, che ha questa sintassi:

```
char* fgets (char *s, int size, FILE *fp);
```

Ad esempio, ho un file contenente i codici fiscali dei miei utenti. Già so che ogni codice fiscale è lungo 16 caratteri, quindi userò la fgets:

```

#include <stdio.h>

#define CF_FILE "cf.txt"

int main(void ) {
    FILE *fp;
    char cf[16];
    int i=1;

    if (!(fp=fopen(USER_FILE,"r")) ) {
        printf ("Errore: impossibile aprire il file %s
                in modalità read-only\n",USER_FILE);
    }
}

```

```

    exit(1);
}

while (!feof(fp)) {
    fgets (cf,sizeof(cf),fp);
    printf ("Codice fiscale n.:%d: %s\n",i++,cf);
}
}

```

Scrittura di dati in formato binario - fwrite

Quelle che abbiamo visto finora sono funzioni per la gestione di file di testo, ovvero funzioni che scrivono su file dati sotto forma di caratteri ASCII. A volte però è molto più comodo gestire file in modalità binaria, ad esempio per file contenenti dati di tipo strutturato, e quindi di dimensione fissata, poiché per quanto grande possa essere il dato strutturato da gestire queste funzioni consentono di gestirlo in una sola lettura e in una sola scrittura.

Per la scrittura di dati binari su file si usa la funzione `fwrite`, che ha questa sintassi:

```
size_t fwrite (void *ptr, size_t size, size_t blocks, FILE *fp);
```

dove `*ptr` identifica la locazione di memoria dalla quale prendere i dati da scrivere su file (può identificare una stringa, un intero, un array...), `size` la dimensione della zona di memoria da scrivere su file, `blocks` il numero di blocchi da scrivere su file (in genere 1) e `*fp` è puntatore a nostro file. `fwrite` ritorna un valore > 0 , che identifica il numero di byte scritti, quando la scrittura va a buon fine, -1 in caso contrario.

Esempio di utilizzo:

```

#include <stdio.h>
#include <stdlib.h>

#define USER_FILE          "user.dat"

typedef struct {
    char user[30];
    char pass[30];
    char email[50];
    int age;
} user;

main() {
    FILE *fp;
    user u;

    if (!(fp=fopen(USER_FILE,"a"))) {
        printf ("Errore: impossibile aprire il file %s
                in modalità append\n",USER_FILE);
        exit(1);
    }
}

```

```

printf ("===Inserimento di un nuovo utente===\n\n");

printf ("Username: ");
scanf ("%s",u.user);

printf ("Password: ");
scanf ("%s",u.pass);

printf ("Email: ");
scanf ("%s",u.email);

printf ("Età: ");
scanf ("%d",&u.age);

// Scrivo i dati su file
if (fwrite (&u, sizeof(u), 1, fp)>0)
    printf ("Dati scritti con successo sul file!\n");
else
    printf ("Errore nella scrittura dei dati su file\n");

fclose (fp);
}

```

Lettura di dati in formato binario - fread

Per la lettura si ricorre invece alla funzione fread, che ha una sintassi molto simile:

```
size_t fread (void *ptr, size_t size, size_t blocks, FILE *fp);
```

Esempio di utilizzo:

```

#include <stdio.h>
#include <stdlib.h>

#define USER_FILE          "user.dat"

typedef struct {
    char user[30];
    char pass[30];
    char email[50];
    int age;
} user;

int main(void) {
    FILE *fp;
    user u;
    int i=0;

    if (!(fp=fopen(USER_FILE,"r"))) {
        printf ("Errore: impossibile aprire il file %s
                in modalità read-only\n",USER_FILE);
    }
}

```

```

    exit(1);
}

while (fread(&u,sizeof(u),1,fp)>0) {
    printf ("Username: %s\n",u.user);
    printf ("Password: %s\n",u.pass);
    printf ("Email: %s\n",u.email);
    printf ("Età: %d\n\n",u.age);
    i++;
}

printf ("Utenti letti nel file: %d\n",i);
fclose (fp);
}

```

Posizionamento all'intero di un file - fseek e ftell

Vediamo ora altre due funzioni indispensabili per il posizionamento all'interno di un file.

Un file è un'entità software memorizzata su un dispositivo ad accesso diretto, come un hard disk o una chiave USB, e in quanto tale è possibile accedere ad esso in qualsiasi punto dopo l'apertura. Ciò è possibile tramite la funzione `fseek`:

*int fseek (FILE *fp, int offset, int whence);*

dove `*fp` è il puntatore al file in cui ci si vuole spostare, `offset` una variabile intera che rappresenta lo spostamento in byte all'interno del file (può essere positiva o anche negativa, nel caso di spostamenti all'indietro) e `whence` rappresenta il punto da prendere come riferimento nello spostamento. In `stdio.h` vengono definiti 3 tipi di `whence`:

- `SEEK_SET` (corrispondente al valore 0), che rappresenta l'inizio del file
- `SEEK_CUR` (corrispondente al valore 1), che rappresenta la posizione corrente all'interno del file
- `SEEK_END` (corrispondente al valore 2), che rappresenta la fine del file

Ad esempio, se come secondo argomento della funzione passo 3 e come terzo argomento `SEEK_CUR`, mi sposterò avanti di 3 byte a partire dalla posizione attuale all'interno del file.

C'è poi la funzione `ftell`:

*int ftell (FILE *fp);*

che non fa altro che ritornare la posizione attuale all'interno del file puntato da `fp` (ovvero il numero di byte a cui si trova il puntatore a partire dall'inizio del file).

Esempio pratico: un programmino per la ricerca di una parola all'interno di un file

```

#include <stdio.h>
#include <stdlib.h>

```

```

#include <string.h>

#define MY_FILE "file_to_search.txt"

main() {
    FILE *fp;
    char s[100];
    char *buff;
    int dim;
    int i=0;

    if (!(fp=fopen(MY_FILE,"r"))) {
        printf ("Errore nella lettura dal file %s\n",MY_FILE);
        exit(1);
    }

    printf ("Parola da cercare all'interno del file %s:",
            MY_FILE);
    scanf ("%s",s);

    dim=strlen(s);
    buff = (char*) malloc(dim*sizeof(char));

    while (!feof(fp)) {
        fscanf (fp,"%s",buff);

        if (!strcmp(s,buff)) {
            printf ("Parola trovata a %d byte dall'inizio\n",
                    ftell(fp)-dim);
            i++;
        }

        /* Mi posiziono indietro nel file di dim+1 caratteri
         * a partire dalla posizione corrente
         */
        fseek (fp,-dim+1,SEEK_CUR);
    }

    printf ("%d occorrenze di %s trovate nel file\n",i,s);
}

```

Prendere parametri da riga di comando

.

Talvolta è molto comodo passare alle nostre applicazioni dei parametri da riga da comando. L'esempio più classico è quello del comando `ls` su sistemi Unix (corrispettivo di `dir` su DOS), che può prendere come parametro la directory di cui si vuole visualizzare il contenuto:

```
ls /home
--- Qui viene stampato il contenuto di /home
```

Si può fare una cosa del genere anche nelle nostre applicazioni. Il metodo standard è quello di richiamare il `main()` del nostro programma con due parametri aggiuntivi:

- **argc** - di tipo `int`
- **argv** - un array di stringhe

Gli argomenti passati alla nostra applicazione verranno piazzati nell'array di stringhe `argv`, mentre invece il numero di argomenti passati sarà indicato dall'intero `argc`:

```
main (int argc, char **argv)
```

Il primo elemento del vettore di stringhe `argv` (`argv[0]`) conterrà sempre il nome dell'applicazione in esecuzione, e quindi `argc` sarà sempre almeno uguale a 1 (in quanto `argv` conterrà sempre almeno un valore). Da `argv[1]` in poi verranno indicati gli argomenti aggiuntivi passati alla nostra applicazione. Esempio pratico:

```
#include <stdio.h>

main (int argc, char **argv) {
    int i;

    printf ("In esecuzione: %s\n",argv[0]);

    // Stampo tutti gli argomenti passati al programma
    for (i=1; i<argc; i++)
        printf ("Argomento n.%d: %s\n",i,argv[i]);
}
```


Libreria math.h

.

Includendo nel proprio codice l'header math.h è possibile utilizzare svariate funzioni e costanti matematiche. Ecco le principali:

Funzioni trigonometriche

- *cos* Calcola il coseno di un numero reale (espresso in radianti)
- *sin* Calcola il seno di un numero reale (espresso in radianti)
- *tan* Calcola la tangente di un numero reale (espresso in radianti)
- *acos* Calcola l'arcocoseno di un numero reale
- *asin* Calcola l'arcoseno di un numero reale
- *atan* Calcola l'arcotangente di un numero reale

Funzioni iperboliche

- *cosh* Calcola il coseno iperbolico di un numero reale
- *sinh* Calcola il seno iperbolico di un numero reale
- *tanh* Calcola la tangente iperbolica di un numero reale

Funzioni esponenziali e logaritmiche

- *exp* Calcola l'esponenziale di un numero reale
- *log* Calcola il logaritmo in base e di un numero reale
- *log10* Calcola il logaritmo in base 10 di un numero reale

Potenze e radici

- *pow* Calcola una potenza. Prende come primo argomento la base e come secondo l'esponente
- *sqrt* Calcola la radice quadrata di un numero reale

Arrotondamento e valore assoluto

- *ceil* Approssima per eccesso un numero reale al numero intero più vicino
- *abs* Calcola il valore assoluto di un numero reale
- *floor* Approssima per difetto un numero reale al numero intero più vicino

Costanti

L'header `math.h` mette anche a disposizione del programmatore alcune costanti matematiche di uso comune con un numero notevole di cifre significative dopo la virgola, senza che ci sia bisogno di definirle di volta in volta. Tra queste il *pi greco* (`M_PI`) e il *numero di Nepero e* (`M_E`).

Generazione di numeri pseudocasuali

In C è possibile generare numeri pseudocasuali in modo relativamente semplice, a patto che si includa l'header `stdlib.h`. Si comincia inizializzando il seme dei numeri casuali tramite la funzione `srand()`. In genere si usa come variabile di inizializzazione del seme la data locale:

```
#include <stdlib.h>
#include <time.h>

...

srand((unsigned) time(NULL));
```

Una volta inizializzato il seme uso la funzione `rand()` per ottenere un numero pseudocasuale. Tale funzione ritorna però numeri estremamente grandi. Per restringere l'intervallo possibile dei numeri pseudocasuali che voglio generare basta calcolarne uno con `rand()` e poi calcolarne il modulo della divisione per il numero più alto dell'intervallo che voglio ottenere. Ad esempio, se voglio ottenere numeri pseudocasuali in un intervallo da 0 a 9 basterà

```
int rnd=rand()%10;
```

Libreria `time.h`

.

La libreria `time.h` è dedicata alla gestione della data e dell'ora. Comprende funzioni di tre tipologie: *tempi assoluti*, rappresentano data e ora nel calendario gregoriano; *tempi locali*, nel fuso orario specifico; *variazioni dei tempi locali*, specificano una temporanea modifica dei tempi locali, ad esempio l'introduzione dell'ora legale. Contiene le dichiarazioni della funzione `time()`, che ritorna l'ora corrente, e la funzione `clock()` che restituisce la quantità di tempo di CPU impiegata dal programma.

`time_t`

Il tipo `time_t`, definito in `time.h`, non è altro che un *long int* addibito al compito di memorizzare ora e date misurate in numero di secondi trascorsi dalla mezzanotte del 1° gennaio 1970, ora di Greenwich. Bisogna ammettere che è un modo un po' bislacco di misurare il tempo, ma è così perché così si è misurato il tempo sui sistemi Unix. Tuttavia, nonostante possa sembrare un modo strano di rappresentare il tempo, questa rappresentazione è estremamente utile per fare confronti tra date che, essendo tutte rappresentate in questo modo, si riducono a semplici confronti tra numeri interi, senza che ci sia bisogno di confrontare giorni, mesi e anni. Ma ci sono anche problemi legati a questa rappresentazione. La rappresentazione del tipo `time_t` infatti è una rappresentazione a 32 bit, che ammette numeri negativi (ovvero numero di secondi prima del 1 gennaio 1970, che consente la rappresentazione di date fino al 1900) e numeri positivi (numero di secondi passati dal 1 gennaio 1970). Il bit più significativo del numero binario identifica il segno (0 per i numeri positivi, 1 per quelli negativi). In questo modo è possibile rappresentare fino a $2^{32}/2$ numeri positivi, ovvero numero di secondi dopo la data per eccellenza, e questo è un problema perché con una tale rappresentazione la data andrà in overflow intorno al 2038 (ovvero, in un certo momento dopo le prime ore del 2038 si arriverà ad un punto in cui la cifra più significativa del numero andrà a 1, quindi le date cominceranno a essere contate dal 1900). Il bug del 2038 è molto conosciuto in ambiente Unix, e per porre rimedio si sta da tempo pensando di migrare ad una rappresentazione della data a 64 bit.

struct tm

tm è una struttura dichiarata sempre in *time.h*, contiene informazioni circa l'ora e la data, questo è il contenuto:

```
struct tm {
int tm_sec    //secondi prima del completamento del minuto
int tm_min    //minuti prima del completamento dell'ora
int tm_hour   //ore dalla mezzanotte
int tm_mday   //giorno del mese
int tm_mon    //mesi passati da gennaio
int tm_year   //anni passati dal 1900
int tm_wday   //giorni passati da Domenica
int tm_yday   //giorni passati dal 1 Gennaio
int tm_isdst  //'unknow' (lol)
};
```

Esempio

Ecco un piccolo programma che ci mostra a schermo ora e data.

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

int main(int argc, char *argv[])
{
    time_t a;
    struct tm *b;

    time(&a);
    b = localtime(&a);
    printf("Ora esatta: %s\n", asctime(b));

    return 0;
}
```

Un altro modo per visualizzare la data attuale senza ricorrere a un membro della struttura *tm* è il seguente:

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

main() {
    time_t ltime = time();
    printf ("%s\n", ctime(&ltime));
}
```

La funzione *ctime* prende l'indirizzo di una variabile di tipo *time_t* inizializzata tramite *time* e stampa il suo valore in formato ASCII. Il formato standard è
Giorno della settimana (3 lettere) Mese (3 lettere) Giorno del mese

(2 cifre) hh:mm:ss Anno (4 cifre)

Un modo per stampare il tempo in un altro formato diverso da quello previsto da `ctime` e `asctime` è quello di usare la funzione `strftime`, che prende come parametri

- Una stringa nella quale salvare la data nel formato che si è scelto
- La dimensione della stringa
- Una stringa di formato (simile a `printf`) nel quale si specifica il formato in cui stampare la data
- Un puntatore a struttura `tm`

Esempio:

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

main() {
    time_t timer=time();
    struct tm *now=localtime(&timer);
    char timebuf[20];

    strftime (timebuf,sizeof(timebuf),"%d/%m/%Y,%T",now);
    printf ("%s\n",timebuf);
}
```

Stampa la data nel formato

gg/MM/aaaa,hh:mm:ss

La funzione `localtime` prende come parametro un puntatore a variabile `time_t` e ritorna una struttura `tm` corrispondente a quel tempo.

Gestione dei file - primitive a basso livello

.

Diverse versioni del C offrono un altro gruppo di funzione per la gestione dei file. Vengono chiamate funzioni di *basso livello*, perché rispetto alle altre corrispondono in modo diretto alle relative funzioni implementate nel kernel del sistema operativo. Vengono impiegate nello sviluppo di applicazioni che necessitano di raggiungere notevoli prestazioni. In questa sede esamineremo le primitive a basso livello per la gestione dei file in ambiente [Unix](#).

Si deve far attenzione a non usare i due tipi di funzione sullo stesso file; le strategie di gestione dei file infatti sono differenti e usarle insieme può generare effetti collaterali sul programma.

File pointer e file descriptor

A differenza delle funzioni d'alto livello definite in `stdio.h`, che utilizzano il concetto di *file pointer*, le funzioni di basso livello fanno uso di un concetto analogo, il *file descriptor* (conosciuto anche come "canale" o "maniglia"). Il file descriptor è un numero intero associato dalla funzione di apertura al file sul quale si opera. Per poter usufruire di queste funzioni è necessario includere nel sorgente i seguenti headers:

- `fcntl.h`
- `sys/types.h`
- `sys/stat.h`

Il file descriptor, a differenza del file pointer definito in `stdio.h` che altro non è che un puntatore alla struttura `FILE`, è un numero intero che identifica in modo univoco il file aperto all'interno della tabella dei files aperti del sistema operativo. In questa tabella i primi 3 numeri (0,1,2) sono riservati ai cosiddetti *descrittori speciali*:

- 0 - *stdin* (Standard Input)
- 1 - *stdout* (Standard Output)
- 2 - *stderr* (Standard Error)

Su un sistema Unix posso quindi scrivere su `stdout` o `stderr` e leggere dati da `stdin` come se fossero normali file, quindi usando le stesse primitive (*everything is a file!*, è un motto comune tra i sistemisti Unix). Se apro un altro file sul mio sistema Unix tale file assumerà quindi un identificatore pari a 3 nella tabella dei file aperti, se ne apro un altro ancora avrà un identificatore 4 e così via.

open

La funzione di apertura si chiama *open*, ecco un esempio:

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

main()
{
    int fd;

    fd = open("nomefile", O_WRONLY);
    ...
}
```

Questa funzione associa *fd* (file descriptor) a *nomefile* e lo apre in modalità di sola scrittura.

La *open* ritorna un valore intero, che è negativo nel caso in cui si è verificato un errore, ad esempio il file non esiste o non si hanno i diritti di lettura/scrittura.

La sintassi della funzione è questa:

```
int fd, modo, diritti;
...
fd = open("nomefile", modo [diritti]);
```

Modalità di apertura

modo rappresenta la modalità di apertura del file, può essere una o più delle seguenti costanti simboliche (definite in *fcntl.h*):

- *O_RDONLY* apre il file in sola lettura
- *O_WRONLY* apre il file in sola scrittura
- *O_RDWR* apre il file in lettura e scrittura

(Per queste tre costanti simboliche, se il file non esiste la *open* ritorna errore)

- *O_CREAT* crea il file
- *O_TRUNC* distrugge il contenuto del file
- *O_APPEND* tutte le scritture vengono eseguite alla fine del file
- *O_EXCL* Se al momento dell'apertura il file già esiste, la *open* ritorna errore

Per poter specificare più di una modalità di apertura si può usare l'operatore di OR bit a bit, esempio:

```
fd = open("nomefile", O_CREAT | O_WRONLY, 0640);
```

Crea il file e lo apre in modalità sola scrittura.

Permessi

Ora vi starete chiedendo cos'è quel 0640, sono i diritti, o permessi, con i quali il file deve essere creato. Sono codificati con una sintassi simile a quella di Unix, che suddivide gli utenti in tre categorie:

- possessore del file;
- appartiene al gruppo collegato al file;
- non è collegato al file in alcun modo.

Per ogni categoria si possono specificare i permessi tramite la forma ottale, costituita da 3 o 4 cifre comprese tra 0 e 7. Esempio:

0640

Tralasciamo il significato della prima cifra a sinistra, che è opzionale. Ogni cifra è da interpretare come una somma delle prime tre potenze di 2 ($2^0=1$, $2^1=2$, $2^2=4$), ognuna delle quali corrisponde ad un permesso - andando da sinistra verso destra, la seconda rappresenta il proprietario, la terza il gruppo e l'ultima tutti gli altri utenti; la corrispondenza è questa:

- 4 permesso di lettura
- 2 permesso di scrittura
- 1 permesso di esecuzione
- 0 nessun permesso

Dunque per ottenere un permesso di lettura e scrittura non occorre far altro che sommare il permesso di lettura a quello di scrittura ($4+2=6$) e così via.

Un altro modo di vedere i permessi Unix di un file è tramite la rappresentazione binaria. I permessi Unix visti sopra non sono altro che una rappresentazione in modo ottale di un numero binario che se visto fa capire al volo quali sono i permessi su un particolare file. Ecco come funziona:

```
U   G   O
rwx rwx rwx
110 100 000
```

In questo caso l'utente (U) ha permessi di lettura e scrittura sul file. Il gruppo (G) ha solo i permessi di lettura. Gli altri utenti non hanno alcun permesso. Se convertiamo ogni gruppetto di 3 cifre in ottale otteniamo 0640, che è effettivamente il permesso che vogliamo.

close

La funzione *close* serve a chiudere un file descriptor aperto dalla *open*:

```
int fd;
...
```



```
close(fd);
```

read e write

Le operazioni di lettura e scrittura sul file, utilizzando i file descriptor, si possono effettuare usando le primitive *read* e *write*.

Esempio di utilizzo di *read*:

```
char buf[100];
int dimensione;
int fd;
int n;
...
dimensione = 100;
n = read(fd, buf, dimensione);
```

fd rappresenta il file descriptor da dove si desidera leggere, *buf* è il vettore che conterrà i dati letti e *dimensione* è la dimensione in byte del vettore.

Il valore di ritorno indica il numero di byte letti da *fd*; questo valore può essere inferiore al valore di *buf*, succede quando il puntatore raggiunge la fine del file; un valore di ritorno uguale a 0 indica la fine del file, mentre invece un valore minore di 0 indica un errore in lettura.

Esempio di utilizzo di *write*:

```
char buf[100];
int dimensione = 100;
int n, fd;
...
n = write(fd, buf, dimensione);
```

fd è il file descriptor da dove si legge, *buf* è il vettore che contiene i dati da scrivere e *dimensione* è la dimensione in byte dei dati da scrivere. Il valore di ritorno della *write* indica il numero di byte scritti sul file; questo valore può essere inferiore alla dimensione nel caso in cui il file abbia superato la massima dimensione ammessa, o inferiore di 0 in caso di errore.

Esempio pratico

Ecco un possibile esempio di lettura tramite le primitive appena viste dei contenuti di un file passato via riga di comando alla nostra applicazione:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
```

```

main (int argc, char **argv) {
    int fd;
    char buff;

    // Controllo se al programma è stato passato almeno un argomento
    if (argc==1) {
        printf ("Uso: %s <file>\n",argv[0]);
        exit(1);
    }

    // Provo ad aprire il file passato
    if ((fd=open(argv[1],O_RDONLY))<0) {
        printf ("Errore nell'apertura di %s\n",argv[1]);
        exit(2);
    }

    // Finché ci sono caratteri da leggere li leggo tramite read...
    while (read (fd,buff,sizeof(buff))>0)
        // ...e li scrivo su stdout tramite write
        // Notate che per la scrittura posso anche usare la write
        // usando come file descriptor 1, che identifica lo stdout
        write (1,buff,sizeof(buff));

    // Chiudo il file
    close (fd);
}

```

lseek

Come per i file pointer, esiste una funzione che consente di muovere il puntatore al file, per i file descriptor si chiama lseek (per i file pointer era fseek). Esempio:

```

long offset;
long n;
int start;
int fd;
...
offset = lseek(fd, n, mode);

```

dove, come sempre, fd è il file descriptor sul quale si muoverà il puntatore; n è il numero di byte che copre lo spostamento (se negativo lo spostamento avviene all'indietro anziché in avanti); mode invece indica la posizione da quale iniziare a muovere il puntatore: se vale 0 ci si deve muovere dall'inizio del file, se vale 1 dalla posizione corrente, mentre se vale 2 a partire dalla fine del file. Il valore di ritorno di lseek contiene la posizione corrente del puntatore (dopo lo spostamento ovviamente). Quindi:

```

lseek(fd, 0L, 1) restituisce la posizione corrente
lseek(fd, 0L, 2) restituisce la dimensione del file in byte

```

Redirezione

La redirezione è qualcosa che ad alto livello, dalla nostra shell [Unix](#), si traduce in qualcosa del tipo

```
./nome_eseguibile > mio_file.txt
```

Ovvero non stampo l'output dell'eseguibile su stdout o stderr, come sarebbe previsto, ma lo re-direziono su un secondo file, magari un file di log. Come si traduce questa caratteristica a basso livello? Semplice. Abbiamo visto che stdin, stdout e stderr sono visti a basso livello come dei semplici file con dei descrittori speciali (rispettivamente 0, 1 e 2). Posso chiudere, ad esempio, il descrittore 1 (stdout) e fare in modo che venga sostituito da un descrittore arbitrario, che in questo caso sarà il descrittore al nostro file di log. Per chiudere il descrittore userò la primitiva già vista `close()`, mentre invece per fare in modo che il descrittore del mio file di log sovrascriva il descrittore di stdout userò la primitiva `dup()` (*duplicate*), che prende come unico argomento il descrittore del mio file e lo copia sul primo descrittore disponibile. In questo caso il primo descrittore disponibile è quello di stdout, lasciato vuoto dalla chiusura di 1, e quindi qualsiasi testo che è indirizzato verso stdout verrà re-indirizzato verso il mio file arbitrario. Esempio pratico:

```
#define MSG "Hello\n"

main() {
    write (1,MSG,sizeof(MSG));
}
```

Questo codice effettuerà come prevedibile la stampa di un semplice messaggio su stdout. Effettuando la redirezione di stdout su un file di log arbitrario diventa:

```
#define MSG "Hello\n"
#define ERR "Impossibile aprire il file di log\n"

main() {
    char *log="mylog.txt";
    int fd;

    if ((fd=open(log,0_WRONLY)<0) {
        write (2,ERR,sizeof(ERR));
        exit(-1);
    }

    // Chiudo stdout
    close(1);

    // Duplico il mio descrittore del log
    // che andrà a sovrascrivere stdout
    dup(fd);

    // A questo punto tutto ciò che doveva finire
```

```

// su stdout verrà re-direzionato sul mio log
write (1,MSG,sizeof(MSG));
close(fd);
}

```

Gestione del filesystem a basso livello

I kernel Unix mettono a disposizione del programmatore anche delle primitive per la gestione del filesystem a basso livello, quali cancellazione e rinominazione di files.

Per rinominare un file la primitiva è *rename()*, che, come prevedibile, prende come argomenti

- Una stringa che identifica l'attuale nome del file
- Una stringa che identifica il nuovo nome da assegnare

Per la cancellazione la primitiva è *unlink()*, che prende come unico argomento una stringa contenente il nome del file da cancelare.

Gestione delle directory

I kernel Unix mettono a disposizione del programmatore anche primitive per la gestione delle directory. Per modificare la directory in cui opera il programma si usa la primitiva *chdir()*, che prende come unico argomento una stringa contenente il nome della directory in cui spostarsi e ovviamente ritorna -1 in caso di errore. Per ottenere invece l'attuale percorso della directory in cui si trova il programma in un certo momento si usa la primitiva *getcwd()*, che prende come argomenti

- Un buffer nel quale salvare il nome della directory corrente
- La sua dimensione

Esempio:

```

#include <stdio.h>
#include <unistd.h>
#include <dirent.h>

main(int argc, char **argv) {
    char dir[MAXNAMLEN];
    char *new_dir;

    if (argc==1) {
        printf ("Uso: %s <dir>\n",argv[0]);
        exit(-1);
    }

    new_dir=argv[1];

    // Cambio la directory corrente
    if (chdir(new_dir)<0) {

```

```

    printf ("Errore - impossibile spostarsi in %s\n",new_dir);
    exit(-2);
}

// Ottengo il nome della directory attuale
// e lo salvo in dir
getcwd (dir,sizeof(dir));

printf ("Directory attuale: %s\n",dir);
}

```

Questo codice semplicemente prende una directory come argomento da riga di comando e prova a spostarsi in quella directory tramite *chdir()*, uscendo in caso di errore. In caso di successo invece salva il percorso della directory corrente in un buffer di dimensione *MAXNAMLEN* (costante definita in *dirent.h* che identifica la dimensione massima che può assumere il nome di una directory) e lo stampa su stdout. In sostanza questo listato fa qualcosa di simile al comando *cd*.

All'interno del file *dirent.h* sono anche definite primitive per la lettura dei file contenuti all'interno di una directory. Ciò che ci serve è un puntatore a directory di tipo *DIR* (non molto diverso in sostanza dal puntatore a file di tipo *FILE* definito in *stdio.h* che abbiamo visto in precedenza) e un puntatore a una struttura di tipo *dirent* che conterrà le informazioni sulla directory. Il campo che ci interessa maggiormente in questo caso della struttura è *d_name*, che conterrà di volta in volta il nome di un file contenuto all'interno della directory. Per l'apertura e la chiusura di un puntatore di tipo *DIR* useremo le primitive *opendir()* e *closedir()*, le cui sintassi non sono molto diverse da quelle di una *fopen* o di una *fclose*:

```

#include <dirent.h>

.....

DIR *dir;
struct dirent *info;

if (!(dir=opendir("nome_dir")))
    // Errore

.....

closedir(dir);

```

Così come *fopen*, *opendir* ritorna NULL nel caso in cui non riesca ad aprire la directory passata come argomento.

Per scannerizzare uno per uno gli elementi della directory si usa la primitiva *readdir()*, che legge le informazioni di tutti i file contenuti nella directory, uno dopo l'altro, e le salva in un puntatore a struttura *dirent*. Quando la lettura è terminata *readdir* ritorna NULL, e si può prendere questa come condizione di stop. A questo punto, con queste nozioni possiamo scrivere un rudimentale programma che si

comporta come il comando *ls* in C, prendendo come parametro da riga di comando il nome della directory di cui si vuole visualizzare il contenuto:

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

main (int argc, char **argv) {
    DIR *dir;
    struct dirent *info;

    if (argc==1) {
        printf ("Uso: %s <dir>\n",argv[0]);
        exit(-1);
    }

    // Apro il descrittore della directory
    if (!(dir=opendir(argv[1]))) {
        printf ("Impossibile aprire la directory %s\n",argv[1]);
        exit(-2);
    }

    // Finché ci sono file all'interno della directory...
    while (info=readdir(dir))
        // ...stampa su stdout il loro nome
        printf ("%s\n",info->d_name);

    // Chiudi la directory
    closedir(dir);
}
```

Socket e connessioni di rete in C

.

Tutte le connessioni di rete, dalle applicazioni per la posta elettronica a quelle per la gestione di una rete aziendale, a livello di protocollo non sono altro che canali di comunicazione tra processi residenti su macchine diverse in collegamento tra loro. La macchina che offre il servizio viene chiamata server, quella che lo richiede client (nel caso particolare della posta elettronica il server sarà il server POP3 o IMAP dal quale scarichiamo i messaggi, il client la nostra applicazione, sia essa Outlook, Thunderbird o Eudora). Il canale che viene creato tra il processo residente sul server e quello residente sul client è il socket, una struttura FIFO (First In First Out) che a livello logico non è molto diverso da una pipe. La differenza sta nel fatto che la pipe è un canale che generalmente mette in comunicazione due processi residenti sulla stessa macchina, mentre il socket è tra due processi residenti su macchine diverse, e inoltre un socket è un canale di comunicazione bidirezionale (sullo stesso socket il client può sia leggere informazioni provenienti dal server sia scrivere informazioni), mentre invece una pipe mette a disposizione un canale per la lettura e uno per la scrittura. A parte queste differenze, a livello concettuale le due strutture per la comunicazione inter-processo sono relativamente simili.

Protocolli TCP e UDP

Negli esempi che prenderemo in esame faremo riferimento al protocollo TCP/IP, lo standard su cui si poggia l'intera infrastruttura di internet. L'altro protocollo (UDP), basato su datagrammi, consente l'invio di pacchetti di dimensioni variabili ed è, per alcune applicazioni, relativamente più veloce, ma non garantisce l'invio di pacchetti ordinati, né l'effettivo recapito a destinazione dei pacchetti stessi (in quanto si tratta di un modello fondamentalmente connectionless oriented, a differenza del TCP che è connection oriented). Prendendo in esame in questa sede applicazioni tipicamente internet-oriented, useremo la famiglia protocollare che in C identifica il dominio di comunicazione internet (definito in `<sys/socket.h>` come `AF_INET`, per il formato di indirizzi IPv4, e `AF_INET6`, per il formato IPv6). Ci sono anche altri domini di socket (dominio Unix, `AF_UNIX`, dominio Novell, `AF_IPX`, e dominio AppleTalk, per macchine Mac, `AF_APPLETALK`), che però non prenderemo in esame in questa sede.

Per gli esempi in C che esamineremo faremo riferimento alle librerie Unix per la gestione dei socket. Con qualche piccola modifica, in ogni caso, il codice qui scritto è operativo anche su sistemi Windows.

Indirizzi IP e endianness

L'indirizzo IP identifica univocamente una macchina all'interno di una rete, e consiste (almeno nella versione 4 del protocollo, versione universalmente usata da anni in tutte le reti) in 4 gruppi di numeri che possono andare da 0 a 255 (in esadecimale 0,...,FF). Un indirizzo IP occupa quindi complessivamente 32 bit (4 byte) in memoria.

Per poter utilizzare indirizzi IP in un'applicazione in C è necessario passare la stringa che corrisponde all'IP alla funzione `inet_addr`, definita in `<arpa/inet.h>`. Esempio:

```
#include <sys/types.h>
#include <arpa/inet.h>

.....

in_addr_t addr;
struct in_addr a;

// Nella mia applicazione, la variabile addr sarà associata all'IP
// di localhost
addr = inet_addr("127.0.0.1")

// Associo al membro s_addr della struttura in_addr la variabile
// appena associata all'indirizzo
a.s_addr=addr;

printf ("Indirizzo IP associato a 0x%x: %s\n",
        addr, inet_ntoa(a));
```

In `<netinet/in.h>` è definita la costante `INADDR_ANY`, che identifica un qualsiasi indirizzo IP (usato nel codice dei server per specificare che l'applicazione può accettare connessioni da qualsiasi indirizzo).

Attenzione, avrete notato l'uso di un membro della struttura `in_addr`. Tale struttura (relativamente scomoda e in sé per sé poco utile, ma preservata nella gestione degli indirizzi in C per una compatibilità con il passato) è deputata a contenere indirizzi di rete, ed è così definita:

```
struct in_addr {
    u_long    s_addr;
}
```

`s_addr` conterrà l'indirizzo ottenuto con `inet_addr`. Noterete poi l'uso della funzione `inet_ntoa` (Network to ASCII), che vuole come parametro un dato di tipo `in_addr`. Tale funzione è necessaria per ottenere una stringa ASCII standard a partire da un indirizzo per un motivo particolare, legato alle convenzioni del protocollo TCP/IP. In tale protocollo, infatti, si usa una convenzione di tipo big endian (ovvero le variabili più grandi di un byte si rappresentano a partire dal byte più significativo). Tale convenzione era in uso anche su altre macchine, come i processori Motorola e i VAX,

ma la maggioranza delle macchine odierne usa lo standard little endian (prima i byte meno significativi e poi a salire quelli più significativi) per rappresentare le informazioni in memoria o nella CPU. Per leggere sulla mia macchina un'informazione passata in formato di rete e viceversa devo quindi fare ricorso a funzioni in grado di passare da una convenzione all'altra. La funzione duale di `inet_ntoa` sarà ovviamente `inet_aton`, che converte una stringa in formato host che rappresenta un IP (quindi con numeri e punti) in formato binario di rete, per poi salvarla all'interno di una struttura `in_addr` passata come parametro alla funzione:

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Esistono anche funzioni per operare conversioni su tipi di dato diversi dalle stringhe, quali `htonl` (da codifica Host Byte Order a codifica Network Byte Order, Long), `htons` (da codifica Host a codifica Network, Short), `ntohl` (da codifica Network a codifica Host, Long) e `ntohs` (da codifica Network a codifica Host, Long).

Porte

Per poter effettuare una connessione non basta un indirizzo IP e il protocollo da usare, è necessario anche specificare la porta dell'host alla quale si desidera collegare il socket, ovvero il servizio da richiedere. In definitiva, quindi, per costruire un socket per la comunicazione tra un client e un server ho bisogno di

- Protocollo per la comunicazione (TCP, UDP)
- Indirizzo IP di destinazione
- Porta su cui effettuare il collegamento

Per poter utilizzare un socket in un programma ho bisogno di far ricorso alle strutture `sockaddr`, definite in `<sys/socket.h>`. La struttura `sockaddr` di riferimento è strutturata in questo modo:

```
struct sockaddr {
    // Famiglia del socket
    short sa_family;

    // Informazioni sul socket
    char sa_data[];
}
```

Nel nostro caso, in cui useremo dei socket per la comunicazione di applicazioni via internet, useremo la struttura `sockaddr_in`, convertendola in `sockaddr`, quando richiesto, tramite operatori di cast:

```
struct sockaddr_in {
    // Flag che identifica la famiglia del socket,
    // in questo caso AF_INET
    short sa_family;

    // Porta
```

```

short sin_port;

// Indirizzo IP, memorizzato in una struttura
// di tipo in_addr
struct      in_addr sin_addr;

// Riempimento di zeri
char  sin_zero[8];
}

```

Ci sono caratteristiche in questa struttura quantomeno curiose e apparentemente obsolete e ridondanti, ma conservate per tradizione e per compatibilità con il passato. In primis il riferimento alla struttura `in_addr` (vista prima) per memorizzare l'indirizzo IP, quando si poteva tranquillamente ricorrere ad una variabile `long`. Il riferimento a questa struttura all'interno di `sockaddr` è uno dei più profondi misteri della tradizione Unix. In secundis, il riempimento della struttura con una stringa (`sin_zero`) che non fa altro che contenere degli zeri, o comunque caratteri spazzatura. Ciò è necessario per rendere la dimensione della struttura pari esattamente a 16 byte, in modo da poter effettuare senza problemi il cast da `sockaddr` a `sockaddr_in` e viceversa (in quanto sono della stessa dimensione).

Inizializzazione dell'indirizzo

Per inizializzare l'indirizzo all'interno della nostra applicazione dovremo quindi far ricorso ad un membro della struttura `sockaddr_in`, specificando al suo interno famiglia protocollare (`AF_INET`), porta e indirizzo IP. Per fare ciò conviene creare una procedura esterna al `main` che faccia il tutto:

```

void addr_init (struct sockaddr_in *addr, int port, long int ip) {
    // Inizializzazione del tipo di indirizzo (internet)
    addr->sin_family=AF_INET;

    // Inizializzazione della porta (da host byte order
    // a network byte order
    addr->sin_port = htons ((u_short) port);

    // Inizializzazione dell'indirizzo (passando per la
    // struttura in_addr
    addr->sin_addr.s_addr=ip;
}

```

Per l'invocazione della procedura ricorreremo a qualcosa del genere:

```

// Porta per il collegamento
#define PORT          3666;

// IP della macchina a cui collegarsi
#define IP            "192.168.1.1"

// Variabile sockaddr_in che identifica la macchina a cui ci

```

```

vogliamo collegare
struct sockaddr_in      server;

.....

addr_init (&server,PORT,inet_addr(IP));

```

Creazione del socket e connessione

Per l'inizializzazione di un socket ricorreremo invece alla primitiva `socket`, che prende come parametri il dominio a cui fare riferimento (nel caso di un'applicazione internet `AF_INET`), il tipo di socket (`SOCK_STREAM` nel caso di TCP, `SOCK_DGRAM` nel caso di UDP) e il protocollo (impostando questo parametro a zero il protocollo viene scelto in modo automatico). La funzione ritorna un valore intero che identifica il socket (socket descriptor) e che verrà usato in seguito per le connessioni, le letture e le scritture, allo stesso modo di un descrittore per file, per processi o per pipe. La funzione ritorna invece -1 nel caso in cui non sia stato possibile creare il socket. Esempio di chiamata:

```

// Descrittore del socket
int sd;

.....

if ((sd=socket(AF_INET,SOCK_STREAM,0))<0) {
    printf ("Impossibile creare un socket TCP/IP\n");
    exit(3);
}

```

Per la chiusura del socket ricorreremo invece alla primitiva `close`, passandogli come parametro il descrittore del nostro socket.

A questo punto è possibile connettersi all'host sfruttando il socket appena creato, usando la primitiva `connect`. Tale primitiva richiede come parametri

- Il descrittore del socket da utilizzare
- Un puntatore a `sockaddr`, contenente le informazioni circa dominio del socket, indirizzo IP di destinazione e porta (l'abbiamo creato in precedenza)
- La dimensione del puntatore a `sockaddr`

La funzione, in modo analogo a `socket`, ritorna -1 nel caso la connessione non sia andata a buon fine. Esempio pratico per il nostro caso:

```

if (connect(sd, (struct sockaddr*) &server, sizeof(struct
sockaddr))<0) {
    printf ("Impossibile collegarsi al server %s sulla porta %d\n",
inet_ntoa(server.sin_addr.s_addr),PORT);
    exit(4);
} else {
    printf ("Connessione effettuata con successo al server %s sulla

```

```
porta %d\n", inet_ntoa(server.sin_addr.s_addr),PORT);  
}
```

In questo caso è richiesto l'operatore di cast esplicito, in quanto in precedenza abbiamo creato una variabile di tipo `sockaddr_in` ma la funzione richiede una variabile di tipo `sockaddr`.

Lettura e scrittura di informazioni sul socket

Creato il canale di comunicazione, per sfruttarlo all'interno della nostra applicazione abbiamo bisogno di scrivere o leggere dati su di esso, in modo da mettere in comunicazione le due macchine. Ciò è possibile grazie alle funzioni `recv` e `send`, rispettivamente per leggere e scrivere sul socket.

La sintassi di queste due funzioni è praticamente uguale. Entrambe prendono 4 parametri:

```
ssize_t recv(int s, void *buf, size_t len, int flags);  
ssize_t send(int s, const void *buf, size_t len, int flags);
```

- Il socket da sfruttare (da cui leggere o su cui scrivere)
- Un puntatore ai dati interessati (una variabile su cui salvare i dati letti o la variabile da scrivere su socket)
- La dimensione dei dati (da leggere o da scrivere)
- Un eventuale flag (si lascia a 0 nella maggior parte dei casi)

Entrambe le funzioni ritornano il numero di byte letti o scritti, quindi si possono fare dei cicli con queste funzioni del tipo “finché ci sono dati da leggere o scrivere su socket, fai una certa cosa” sfruttando il fatto che quando non ci sono più dati le funzioni ritornano zero.

Lato server

Per inizializzare una comunicazione di rete su un client basta questa procedura:

- **addr_init** (inizializzazione della variabile di tipo `sockaddr_in` che identifica l'indirizzo e la porta)
- **socket** (creazione del socket per la comunicazione con il server)
- **connect** (connessione al server sfruttando il socket appena creato)

Su un server sono necessari un paio di passaggi in più. La procedura in genere è questa (non solo in C ma per qualsiasi linguaggio di programmazione):

- **addr_init** (inizializzazione della variabile di tipo `sockaddr_in`)
- **socket** (creazione del socket)
- **bind** (creazione del legame tra il socket appena creato e la variabile `sockaddr_in` che identifica l'indirizzo del server)
- **listen** (mette il server in ascolto per eventuali richieste da parte dei client)

- **accept** (accettazione della connessione da parte di un client)

La sintassi di bind è la seguente:

```
int bind(int sockfd, const struct sockaddr *my_addr, socklen_t
        addrLen);
```

dove *sockfd* è l'identificatore del socket, **my_addr* il puntatore alla variabile di tipo *sockaddr* che identifica l'indirizzo e *addrLen* la lunghezza di tale variabile. La funzione ritorna 0 in caso di successo, -1 in caso di errore.

La sintassi di listen invece è la seguente:

```
int listen(int sockfd, int backlog);
```

dove *sockfd* è il descrittore del socket e *backlog* il numero massimo di connessioni che il server può accettare contemporaneamente. Anche questa funzione ritorna 0 in caso di successo e -1 in caso di errore.

La sintassi di accept infine è la seguente:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t
        *addrLen);
```

dove *sockfd* è il descrittore del socket, **addr* il puntatore alla variabile di tipo *sockaddr* e **addrLen* il puntatore alla sua lunghezza. In caso di successo *accept* ritorna un valore >0 che è il descrittore del socket accettato, mentre ritorna -1 in caso di errore.

Attenzione: la *accept* ritorna un nuovo identificatore di socket, che è il socket da utilizzare da quel momento in poi per le comunicazioni con il client. Inoltre, alla *accept* va passato il puntatore alla variabile *sockaddr* che identifica il client, non quello del server.

Esempio pratico

Bando alle ciance, vediamo ora un semplice codice in C per l'invio di messaggi sulla rete sfruttando i socket TCP che abbiamo appena esaminato. Il server rimane in attesa di messaggi sulla porta 3666 e quando arrivano li scrive su *stdout*, mentre il client si collega al server (il cui indirizzo è passato come parametro da riga di comando) e gli invia un messaggio, passato anch'esso come una lista di parametri da riga di comando.

Codice del client:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/types.h>
```

```

#include <sys/wait.h>
#include <sys/socket.h>
#include <errno.h>

// Porta per la comunicazione
#define PORT          3666

// Inizializzazione della variabile sockaddr_in
void addr_init (struct sockaddr_in *addr, int port, long int
ip) {
    addr->sin_family=AF_INET;
    addr->sin_port = htons ((u_short) port);
    addr->sin_addr.s_addr=ip;
}

main(int argc, char **argv) {
    int i,sd;
    int var1,var2,var3,var4;
    int sock_size=sizeof(struct sockaddr_in);
    int N,status;
    pid_t pid;
    struct sockaddr_in server,client;

    // Controllo che vengano passati almeno due argomenti
    if (argc<3) {
        printf ("%s <server> <msg>\n",argv[0]);
        exit(1);
    }

    // Controllo che l'IP del server passato sia un
indirizzo IPv4 valido
        if (sscanf(argv[1],"%d.%d.%d.
%d",&var1,&var2,&var3,&var4) != 4) {
            printf ("%s non è un indirizzo IPv4
valido\n",argv[1]);
            exit(2);
        }

    // Inizializzazione dell'indirizzo
    addr_init (&server,PORT,inet_addr(argv[1]));

    // Creazione del socket
    if ((sd=socket(AF_INET,SOCK_STREAM,0))<0) {
        printf ("Impossibile creare un socket
TCP/IP\n");
        exit(3);
    }

    // Creazione della connessione
    if (connect(sd, (struct sockaddr*) &server,
sock_size)<0) {
        printf ("Impossibile collegarsi al server %s
sulla porta %d: errore %d\n",

```

```

dr),PORT,errno);
    inet_ntoa(server.sin_addr.s_ad
    exit(4);
}

    printf ("Connessione stabilita con successo con il
server %s sulla porta %d\n",
    inet_ntoa(server.sin_addr.s_addr),
ntohs(server.sin_port));

    // Il numero di parole contenute nel messaggio è pari
ad argc-2,
    // ovvero argc-(nome del programma)-(IP del server)
    N=argc-2;

    // Dico al server che sto per inviargli N stringhe
send (sd, (int*) &N, sizeof(int), 0);

    // Per i che va da i ad argc...
for (i=2; i<argc; i++) {
    // ...N è la lunghezza dell'i-esima stringa
    N=strlen(argv[i]);

    // Dico al server che sto per inviargli una
stringa lunga N caratteri
send (sd,(int*)&N,sizeof(int),0);

    // Invio al server la stringa
send (sd,argv[i],N,0);
    printf ("Stringa %s lunga %d caratteri inviata
con successo al server %s\n",
    argv[i],N,inet_ntoa(server.sin
_addr.s_addr));
}

    // Chiusura della connessione
close(sd);
exit(0);
}

```

Codice del server:

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>

// Porta su cui mettersi in ascolto

```

```

#define PORT                3666

// Numero massimo di connessioni accettabile
#define MAXCONN 5

void addr_init (struct sockaddr_in *addr, int port, long int
ip) {
    addr->sin_family=AF_INET;
    addr->sin_port = htons ((u_short) port);
    addr->sin_addr.s_addr=ip;
}

main() {
    int sd,new_sd;
    struct sockaddr_in server,client;
    int sock_size=sizeof(struct sockaddr_in);
    int pid,status;
    int i,args,N;
    char *buff;

    // Inizializzazione dell'indirizzo
    // Con INADDR_ANY specifico che posso accettare
connessioni da qualsiasi indirizzo
    addr_init (&server,PORT,INADDR_ANY);

    // Creazione del socket
    if ((sd=socket(AF_INET,SOCK_STREAM,0)) < 0) {
        printf ("Impossibile inizializzare il socket
TCP/IP %d\n",
                                getsockname (sd, (struct
sockaddr*) &server, &sock_size));
        exit(1);
    }

    // Lego il socket appena creato all'indirizzo del
server
    if (bind(sd, (struct sockaddr*) &server,
sizeof(server))<0) {
        printf ("Impossibile aprire una connessione
sulla porta %d\n"
                                "La porta potrebbe essere già
in uso da un'altra applicazione\n",PORT);
        exit(2);
    }

    printf ("Server in ascolto sulla porta %d\n",PORT);

    // Metto il server in ascolto
    if (listen(sd,MAXCONN)<0) {
        printf ("Impossibile accettare nuove
connessioni sul socket creato\n");
        exit(3);
    }
}

```



```

    printf ("Server in ascolto - accetta fino a un massimo
di %d connessioni\n",MAXCONN);

    // Accetto connessioni finché ce ne sono
    while (1) {
        // Accetto le connessioni da parte del client
        creando un nuovo socket
        if ((new_sd=accept(sd, (struct sockaddr*)
&client, &sock_size)) < 0) {
            printf ("Impossibile accettare una
connessione dal client %s\n",
            inet_ntoa(client.sin_addr.s_addr));
            exit(4);
        }

        printf ("Connessione stabilita con successo con
il client %s sulla porta %d\n",
            inet_ntoa(client.sin_addr.s_addr), ntohs
(client.sin_port) );

        // Ricevo il numero di messaggi che il client ha
da inviare
        recv (new_sd, (int*) &args, sizeof(int), 0);

        printf ("Stringa ricevuta da %s: ",
            inet_ntoa(client.sin_addr.s_addr));

        // Finché il client ha stringhe da inviare...
        for (i=0; i<args; i++) {
            // ...leggo la dimensione dell'i-esima
stringa
            recv (new_sd, (int*) &N, sizeof(int), 0);

            // Alloco memoria per ricevere la stringa
            buff = (char*) malloc(N*sizeof(char));

            // Ricevo la stringa
            recv (new_sd,buff,N,0);

            // Scrivo su stdout la stringa appena
ricevuta
            printf ("%s ",buff);
        }

        printf ("\n");
    }
}

```

Multiprogrammazione - programmazione multiprocesso e multithread

.

Introduzione ai sistemi multiprogrammati

Ciò che ha fatto la fortuna dei sistemi Unix già negli anni '80 è la propensione di questi ultimi nei confronti della programmazione multiprocesso e, in seguito, anche della programmazione multithread.

Per capire cosa voglia dire la programmazione multiprocesso, faccio un esempio. I sistemi DOS degli anni '80 erano sistemi monoprogrammati, ovvero consentivano alla CPU di eseguire un solo processo per volta. Se quindi volevo effettuare, ad esempio, la copia di una directory in un'altra su uno di questi sistemi, il processo di copia occupava interamente le risorse della CPU, e fino alla terminazione di questo processo all'utente era impossibile avviare nuovi processi.

I sistemi Unix invece hanno adottato vari stratagemmi per consentire, anche ai sistemi con una sola CPU, di eseguire più task in maniera concorrente. Questo si riduce, a livello di sistema, a una segmentazione della memoria in più processi. La CPU può sempre eseguire un solo processo per volta, nel caso dei sistemi monoprocesso, ma i sistemi Unix sono programmati in modo tale che il processo che in un dato momento occupa la CPU non “congeli” le risorse del sistema fino alla sua terminazione, adottando politiche di scheduling (a “rotazione” di processi).

Algoritmi di scheduling

Nel corso degli anni gli algoritmi di scheduling si sono sempre più evoluti, cercando di evitare da una parte problemi quali l'occupazione prolungata della CPU da parte di un solo processo e, dal lato opposto, problemi di starvation (ovvero il congelamento di un processo che, a causa di politiche errate nell'algoritmo di scheduling, quali un'errata gestione della priorità dei processi, non acquisterà mai una priorità sufficiente per essere eseguito, rimanendo per sempre in attesa).

L'algoritmo di scheduling più elementare è quello round-robin, un algoritmo che suddivide il tempo della CPU in quanti uguali. Esempio: ho 3 processi in esecuzione, con i seguenti tempi:

```
task1 = 250ms
task2 = 150ms
task3 = 200ms
```

con una politica round-robin che, mettiamo, suddivide il tempo di utilizzo della CPU in 50ms per ogni task, avrò:

1.task1 occupa la CPU per 50 ms (ovvero, passa dallo status READY in cui si trova prima di andare in esecuzione allo status RUNNING per 50ms, per poi essere fermato dal sistema operativo, passando in status SLEEPING e, dopo un certo intervallo, nuovamente in status READY) 2.task2 occupa la CPU per 50 ms 3.task3 occupa la CPU per 50 ms 4.task1 occupa la CPU per 50 ms 5.task2 occupa la CPU per 50 ms 6.task3 occupa la CPU per 50 ms 7.task1 occupa la CPU per 50 ms 8.task2 occupa la CPU per 50 ms (a questo punto il codice da eseguire all'interno di task2 è terminato) 9.task3 occupa la CPU per 50 ms 10.task1 occupa la CPU per 50 ms 11.task3 occupa la CPU per 50 ms (a questo punto il codice da eseguire all'interno di task3 è terminato) 12.task1 occupa la CPU per 50 ms (a questo punto anche il codice da eseguire all'interno di task3 è terminato)

Quest'algorithmo è semplice da implementare a livello di kernel ed evita problemi di starvation, in quanto non ha una politica predefinita per le priorità di un task. Tuttavia, attraverso quest'algorithmo più il task è grande (ovvero maggiore è il suo tempo di esecuzione), più viene premiato, in quanto può occupare la CPU per un periodo cumulativo di tempo maggiore dei task più piccoli.

Gli algoritmi round-robin, nelle varianti weighted round-robin e deficit round-robin, vengono anche utilizzati per lo scheduling dei pacchetti provenienti da connessioni multiple. Ad esempio, se il sistema riceve dei pacchetti da n fonti f_1, f_2, \dots, f_n , è possibile attraverso algoritmi di questo tipo stabilire per quanto tempo ogni fonte è autorizzata a inviare pacchetti al sistema.

L'altra grande classe di algoritmi di scheduling, ideata per evitare i problemi degli algoritmi RR, sono gli algoritmi a priorità, ideati per evitare che i task che richiedono un tempo di esecuzione maggiore vengano maggiormente premiati, come negli algoritmi RR. Gli algoritmi di questo tipo si dividono a loro volta in

Algoritmi a priorità statica. In questi algoritmi la priorità di un task viene stabilita all'atto della sua creazione, in base alle sue caratteristiche Algoritmi a priorità dinamica. In questi algoritmi la priorità di un task può variare durante l'esecuzione. Questo è utile per i seguenti motivi: Per penalizzare i task che impegnano troppo la CPU Per evitare problemi di starvation (ovvero per evitare che nella coda di esecuzione dei processi non riescano mai ad andare in esecuzione) Per aumentare la priorità di un processo in base al suo tempo di attesa nella coda

Programmazione multiprocesso

Un sistema Unix è fortemente improntato sulla programmazione multiprocesso. In particolare, quando un sistema Unix viene avviato viene anche generato un processo, chiamato `init`, con la priorità massima. Questo processo è alla base di tutti i processi che vengono successivamente generati all'interno del sistema. Le shell altro non sono che processi figli del processo `init`, la procedura di autenticazione attraverso username e password è a sua volta gestita da altri due processi, generalmente generati dalla shell stessa, i processi `login` e `getty`. E ancora, ogni eseguibile avviato nel sistema non fa altro che generare un nuovo processo all'interno della shell che lo ha richiamato, e a sua volta l'eseguibile stesso può generare altri processi (vedremo presto come farlo). Un processo eredita dal processo che lo ha richiamato l'area dati (ovvero le variabili presenti all'interno dello stack dell'eseguibile prima che venisse generato il processo figlio). Ma, una volta che viene mandato in esecuzione, ha una propria area dati (questo vuol dire che le modifiche attuate all'interno della propria area dati non modificano i dati all'interno del processo padre) e, ovviamente, una propria area di codice, che contiene il codice che il processo deve eseguire. Sui sistemi Unix per generare un nuovo processo si utilizza la primitiva `fork()`. Questa primitiva fa le operazioni appena descritte sopra, ovvero genera un nuovo processo, con un proprio PID (Process ID, ovvero un numero che identifica il processo) e copia all'interno della sua area dati l'area dati del processo padre. La primitiva `fork()` ritorna

- 0 nel caso del processo figlio (quindi, se il risultato della `fork()` è 0 so che lì ci devo andare a scrivere il codice che verrà eseguito dal processo figlio)
- un valore > 0 nel caso del processo padre
- -1 se c'è stato un errore (ad esempio, se la tabella dei processi è piena, se non ho abbastanza spazio in memoria per allocare un nuovo processo, se non ho i diritti per creare nuovi processi)

Facciamo un primo esempio di programma concorrente multiprocesso in C:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int pid;
    int status;

    printf ("Sono il processo padre, il mio PID è %d\n", getpid());

    /* Genero un nuovo processo */
    pid = fork();

    if ( pid == -1 ) {
        /* ERRORE! Non è stato possibile creare il nuovo processo */
        printf ("Impossibile creare un nuovo processo\n");
    }
}
```

```

    exit(1);
}

if ( pid == 0 ) {
    /* In questo caso la fork() ha ritornato 0, quindi qui ci scrivo
il codice del figlio */
    printf ("Sono il processo figlio di %d, il mio PID è %d\n",
getppid(), getpid());
    exit(0);
}

if ( pid > 0 ) {
    /* In questo caso la fork() ha ritornato un valore maggiore di
0, quindi qui scrivo il codice del processo padre */
    printf ("Sono il processo padre e ho generato un processo
figlio\n");

    /* Attendo che il processo figlio venga terminato, e salvo il
suo valore di ritorno nella variabile status */
    while ( (pid = wait(&status)) > 0 );
        /* Dal valore di status ricavo il valore di ritorno del
processo figlio */
        status = (status & 0xFF) >> 8;

    printf ("Il processo %d è terminato con status %d\n", pid,
status);
}

return 0;
}

```

Un paio di commenti. Innanzitutto, un processo può conoscere in ogni momento il suo PID e il PID del processo che lo ha generato rispettivamente attraverso le primitive getpid() e getppid() (GET Parent PID). Lo studio dei valori di ritorno della primitiva fork() è già stato fatto precedentemente e commentato nel codice, quindi non sto qui a discuterlo nuovamente. È invece interessante l'uso della primitiva wait(), utilizzata all'interno del codice. Questa primitiva mette il processo padre in attesa finché tutti i processi figli non vengono terminati, ritorna -1 se non ci sono processi figli da attendere o un valore > 0 che rappresenta il PID del processo figlio appena terminato. Come parametro prende invece un puntatore a una variabile int. Su questa variabile viene scritto lo status con cui è terminato il processo figlio (attraverso un return o la primitiva exit()) nel seguente formato (in esadecimale):

0xSS00

dove SS rappresenta lo status (in esadecimale) con cui il programma è terminato (nel nostro caso 0), e le ultime due cifre sono 2 zeri. Questo nel caso in cui il processo è terminato in modo "naturale". Se invece dovesse essere terminato in modo "innaturale", ovvero tramite un segnale da parte del padre, gli zeri e il valore dello status verrebbero invertiti. Per ottenere il valore di ritorno devo quindi ricorrere a uno

stratagemma a “basso livello”. Faccio un AND tra la mia variabile e il numero esadecimale 0xFF00 (in binario 1111 1111 0000 0000), in modo da azzerare eventuali valori diversi da zero nelle due cifre esadecimali meno significative, quindi faccio uno shift a destra di 1 byte del valore attuale, in modo da ritrovarmi con un valore del tipo 0x00SS, che rappresenta lo status autentico ritornato dal processo figlio. La riga

```
while ((pid=wait(&status)>0);
```

dice quindi al processo padre “finché la primitiva wait() ritorna un valore maggiore di zero, ovvero finché ci sono processi da attendere, salva questo valore nella variabile pid, quindi salva il valore dello status nella variabile status”. Nel caso il valore di ritorno dei miei processi non mi interessi più di tanto, posso scrivere

```
while ((pid=wait((int*) 0)>0);
```

Vediamo ora un piccolo esempio di programma al quale vengono passati due argomenti, rappresentanti due nomi di file, e che genera due processi figli, ognuno dei quali legge un carattere dal file ad esso associato e lo riporta su standard output. Ogni processo figlio ritorna al padre il numero di caratteri letti all'interno del file. (N.B.: in questo esempio ho utilizzato le primitive a basso livello del kernel Unix, ovvero open, read, write e close, per l'apertura/lettura/scrittura/chiusura di un file, e non le funzioni ad alto livello specificate in stdio.h, proprio perché voglio creare un programma ottimizzato al 100% per sistemi Unix, e suppongo che il lettore sia familiare con queste primitive. In caso contrario, si possono visionare le pagine di manuale Unix associate, o la documentazione presente su internet)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
    /* Descrittore dei file */
    int fd;
    int i, pid, status;
    /* Numero di caratteri letti */
    int N = 0;
    /* Buffer in cui verrà salvato il carattere letto */
    char buff[1];

    if ( argc < 3 ) {
        printf ("Errore nel numero di argomenti passati\n");
        exit(1);
    }

    /* Creo i due processi */
    for ( i = 0; i < 2; i++ ) {
        pid = fork();
```

```

/* Errore */
if ( pid == -1 ) {
    printf ("Errore nella creazione del processo figlio\n");
    exit(2);
}

/* Codice del figlio */
if ( pid == 0 ) {
    /* Provo ad aprire il file associato al processo */
    if ( (fd = open(argv[i+1], O_RDONLY)) < 0 ) {
        printf ("Errore: impossibile leggere il file %s\n",
argv[i+1]);
        exit(3);
    }

    printf ("Contenuto del file %s:\n", argv[i+1]);

    /* Finché ci sono caratteri da leggere all'interno del
file, li riporto su stdout */
    while ( read(fd,buff,1) > 0 ) {
        printf ("%c", buff[1]);
        N++;
    }

    close(fd);

    /* Ritorno il numero di caratteri letti */
    exit(N);
}

/* Codice del processo padre */
if ( pid > 0 ) {
    while( (pid = wait(&status)) > 0 ) {
        status = (status & 0xFF) >> 8;
        printf ("Il processo %d è terminato e ha letto %d
caratteri dal file\n", pid, status);
    }
}

return 0;
}

```

Comunicazione tra processi. Concetto di pipe

Due processi possono comunicare e scambiarsi dati tra loro attraverso un meccanismo di astrazione chiamato pipe. In italiano potremmo tradurlo come “tubo”, e questa parola rende molto bene l’idea. Una pipe è una struttura astratta per la comunicazione tra due o più processi che si può schematizzare proprio come una

tubatura. Dal punto di vista logico-informatico è una struttura FIFO (First In First Out); questo vuol dire che, se un processo scrive sulla pipe e un altro legge i dati scritti, quest'ultimo legge i dati nell'ordine preciso in cui sono stati scritti. Dal punto di vista di sistema, una pipe viene descritta da un array di due interi: il primo valore dell'array identifica il canale di lettura della pipe, il secondo valore identifica quello di scrittura. Su un sistema Unix per inizializzare una pipe uso la primitiva pipe(), primitiva che ritorna un valore ≥ 0 nel caso in cui la pipe è creata con successo, -1 in caso contrario, e prende come parametro l'identificatore della pipe. Piccolo esempio di utilizzo:

```
/* Identifico il tipo pipe_t (pipe type) come un array di due interi
*/
typedef int pipe_t[2];

...

pipe_t pp;

if (pipe(pp)<0) { Errore! }

/* D'ora in avanti userò il canale pp[0] per leggere dalla pipe,
pp[1] per scrivere sulla pipe */
```

Ovviamente, se provo a scrivere sul canale di lettura della pipe o viceversa ottengo un errore di broken pipe, in quanto sto tentando di eseguire un'operazione non consentita. Vediamo ora un esempio più corposo, in cui un processo padre inizializza una pipe e crea un processo figlio. Il processo figlio prende da stdin una stringa, di lunghezza massima N, inserita dall'utente e la scrive sulla pipe. Il processo padre attende che il figlio termini e scrive la stringa su stdout, leggendola dal canale di lettura della pipe. (N.B.: per leggere, scrivere o chiudere una pipe utilizzo sempre le primitive read, write e close, esattamente le stesse primitive che userei per un file o per un socket).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

/* Massima lunghezza dell'input inserito */
#define N      100

typedef int pipe_t[2];

int main(int argc, char *argv[])
{
    pipe_t pp;
    char buff[N];

    /* Creazione della pipe */
```



```

    if (pipe(pp) < 0) {
        printf ("Errore nella creazione della pipe\n");
        exit(1);
    }

    /* Creazione di un processo figlio */
    switch (fork()) {
        case -1: printf ("Impossibile creare un processo
figlio\n");
                exit(2);
                break;
        /* Codice del figlio */
        case 0: /* Chiudo il canale di lettura della pipe, in
quanto il processo figlio deve
                solo scrivere sulla pipe e il canale di lettura
non mi interessa */
                close(pp[0]);

                /* Chiedo all'utente di inserire una stringa */
                printf ("Stringa da inviare sulla pipe: ");
                fgets(buff, N, stdin);
                buff[strlen(buff)]='\0';

                /* Scrivo la stringa appena letta sulla pipe */
                if (write(pp[1], buff, N) < 0) {
                    printf ("Errore nella scrittura su pipe\n");
                    exit(3);
                }

                close(pp[1]);
                exit(0);
                break;
        /* Codice del padre */
        default: /* Chiudo il canale di scrittura dal lato del
padre, dato che devo solo leggere dalla pipe */
                close(pp[1]);

                printf ("Aspetto che il figlio venga terminato...\n");

                wait( (int*) 0 );

                /* Una volta che il figlio è terminato, leggo la
stringa inserita dalla pipe */
                if (read(pp[0], buff, N) < 0) {
                    printf ("Errore nella lettura da pipe\n");
                    exit(1);
                }

                printf ("Il processo figlio ha scritto %s sulla
pipe\n", buff);

                exit(0);
                break;
    }
}

```

```

    return 0;
}

```

È possibile anche effettuare la ridirezione di un certo canale su una pipe. Ricordiamo che in un sistema Unix `stdin`, `stdout` e `stderr` non sono altro che descrittore di file “speciali”, identificati rispettivamente dai valori 0, 1 e 2. Quindi posso chiudere uno di questi canali e ridirigere il traffico diretto da o verso uno di questi canali su una pipe, così come potrei fare la ridirezione su file. Esempio:

```

...
typedef int pipe_t[2];

pipe_t pp;

...

close(1);          /* Chiudo stdout */
dup(pp[1]);        /* Duplico il canale di scrittura della pipe, che
acquista il primo canale disponibile. */
/* Poiché ho appena chiuso il canale stdout, tutto il traffico
diretto su stdout verrà
* ridiretto sulla pipe. */

```

Questo meccanismo, che ora abbiamo visto implementato a basso livello, viene implementato ad alto livello dai comandi di pipe della shell. Ad esempio se do un comando del tipo `ps ax | grep init`, non fa altro che eseguire il comando `ps`. Il canale di output di questo comando viene chiuso, e viene ridiretto su una pipe costruita per la comunicazione tra `ps` e `grep`.

Interruzione di un processo. Concetto di segnale

Il processo padre può terminare, uccidere un processo figlio o imporgli l'esecuzione di un codice arbitrario in un certo momento attraverso il meccanismo dei segnali. Questa è la lista dei segnali standard su un sistema Unix, visibile anche dando il comando `man 7 signal`:

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal

SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

I segnali si installano con la primitiva `signal()` (standard SystemV, quello più utilizzato) o `sigset()` (standard BSD, meno utilizzato). Entrambe le primitive prendono come primo argomento il segnale associato (uno di quelli presenti nella lista), come secondo argomento una funzione di tipo `void` che prende un parametro di tipo `int` (che rappresenta il numero del segnale ricevuto); questa è la funzione che verrà richiamata quando viene lanciato un dato segnale. Il segnale viene invece “lanciato” con la primitiva `kill()`, una primitiva che prende come primo parametro il PID del processo che deve ricevere il segnale, come secondo il segnale da inviare. Esempio:

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

void foo(int sig) {
    printf ("Ricevuto segnale %d. Terminazione del processo...\n",
sig);
    signal(SIGTERM,foo);
}

void do_nothing(int sig) {
    signal(SIGUSR1,do_nothing);
}

main() {
    int pid;

    // Installazione dei segnali
    signal(SIGTERM,foo);
    signal(SIGUSR1,do_nothing);

    pid=fork();

    switch(pid) {
        case -1:
            printf ("Errore nella creazione del processo\n");
            exit(1);
```

```

        break;

// Figlio
case 0:
    printf ("Sono il processo %d, generato da %d, e attendo un
segnale da parte di mio padre\n", getpid(), getppid());

        // Invio al processo padre il segnale SIGUSR1, un segnale
"personalizzato"
        kill(getppid(), SIGUSR1);

        // Pongo il processo in attesa di un segnale attraverso la
primitiva pause()
        pause();
        exit(0);
        break;

// Padre
default:
    // Mi metto in attesa di un segnale
    pause();

    // Una volta ricevuto il segnale SIGUSR1 da parte del figlio,
mando al figlio il segnale SIGTERM
    kill(pid, SIGTERM);
    exit(0);
    break;
}
}

```

Programmazione multithread

Il concetto di thread, pur essendo operativamente molto simile a quello di processo, è in sostanza un concetto diverso. Fondamentalmente, l'inizializzazione di un nuovo processo è sempre qualcosa di oneroso per il sistema, in quanto un processo ha una sua area di memoria (ovvero una sua area di codice, una sua area di dati e un suo stack) e un suo PID che lo identifica all'interno della tabella dei processi, e alla sua inizializzazione il sistema operativo dovrà provvedere al nuovo processo le risorse di memoria richieste. Il thread invece lo possiamo vedere come un "mini-processo" (per usare una terminologia un po' grossolana ma che rende bene l'idea) che può essere richiamato all'interno di un processo stesso. Il thread condivide l'area di memoria con lo stesso processo chiamante (ovvero condivide con il processo chiamante gli stessi dati e la stessa area di stack, il che vuol dire che una modifica sulle variabili operata da un thread è visibile da tutti gli altri thread del processo stesso). Il vantaggio principale della programmazione multithread è la maggiore velocità di inizializzazione e di esecuzione di un thread rispetto a quella di un processo, a costo di una minore indipendenza in fatto di memoria condivisa tra i thread di un processo stesso. Per ricorrere alla programmazione multithread in C in ambiente Unix useremo

la libreria pthread (inclusa di default in molte installazioni Unix), e compileremo i sorgenti con l'opzione -lpthread. Per creare un nuovo thread useremo la funzione pthread_create(), che prende come parametri un puntatore all'identificatore del thread (una variabile di tipo pthread_t, tipo definito nell'header sys/types.h), gli attributi del thread creato (generalmente NULL), il puntatore alla funzione contenente il codice che verrà eseguito dal thread (generalmente una funzione di tipo void* che prende un argomento di tipo void*) e un array contenente gli argomenti da passare alla funzione. Userò invece la funzione pthread_exit() per terminare l'esecuzione di un thread (questa funzione prende come parametro il valore da ritornare al processo chiamante) e pthread_join() per porre il processo chiamante in attesa finché il thread creato non viene terminato (questa funzione prende come argomenti l'identificatore del thread e un puntatore alla variabile in cui verrà salvato il valore di ritorno del thread).

Esempio:

```
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>

// Funzione che verrà eseguita dal thread
void* start(void* arg) {
    printf ("Sono un thread richiamato dal processo padre\n");
    pthread_exit(0);
}

main() {
    // Identificatore del thread
    pthread_t t;
    int status;

    if (pthread_create(&t, NULL, start, NULL) != 0) {
        printf ("Errore nella creazione del nuovo thread\n");
        exit(1);
    }

    // Attendo che il thread venga terminato
    pthread_join(t, &status);

    printf ("Il thread a 0x%x è terminato con status %d\n", &t,
status);
}
```

Vediamo ora come passare degli argomenti alla funzione del thread:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>

// Funzione che verrà eseguita dal thread
```

```

void* start(void* arg) {
    // L'argomento passato è sottoforma di dato void, ovvero un dato
    grezzo.
    // Lo converto in int attraverso un operatore di cast
    int *my_arg = (int) arg;

    printf ("Sono un thread generato dal processo padre. Mi è stato
    passato come argomento %d\n", (*x));
    pthread_exit(0);
}

main(int argc, char **argv) {
    pthread_t t;
    int status;
    int arg[1];

    if (argc<2) {
        printf ("Passami almeno un parametro\n");
        exit(1);
    }

    arg[0]=atoi(argv[1]);

    if (pthread_create(&t, NULL, start, arg) != 0) {
        printf ("Errore nella creazione del nuovo thread\n");
        exit(1);
    }

    // Attendo che il thread venga terminato
    pthread_join(t, &status);

    printf ("Il thread a 0x%x è terminato con status %d\n", &t,
    status);
}

```

Programmazione della porta parallela in C



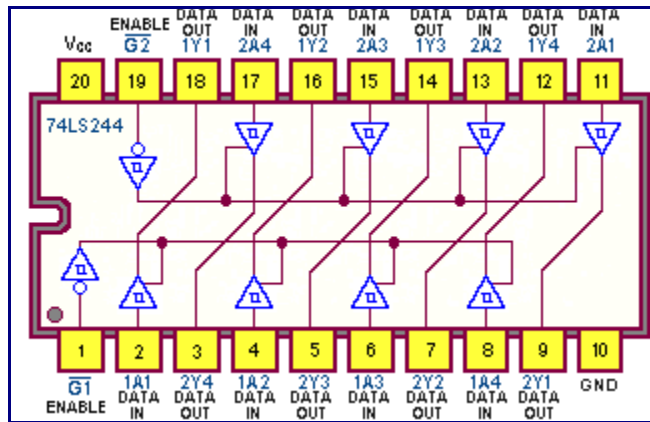
Retro di un moderno PC. In viola, la porta parallela

La porta parallela è una delle principali interfacce I/O su un calcolatore. Inizialmente usata per connettere la stampante al computer (oggi su molte macchine moderne questa porta non è neanche più presente in quanto la maggior parte delle stampanti al giorno d'oggi usano un'interfaccia USB), la porta parallela è in seguito diventata un'interfaccia estremamente utilizzata da elettronici e informatici per pilotare tramite il calcolatore dispositivi *self-made*, in virtù dell'estrema facilità di programmazione di quest'interfaccia.

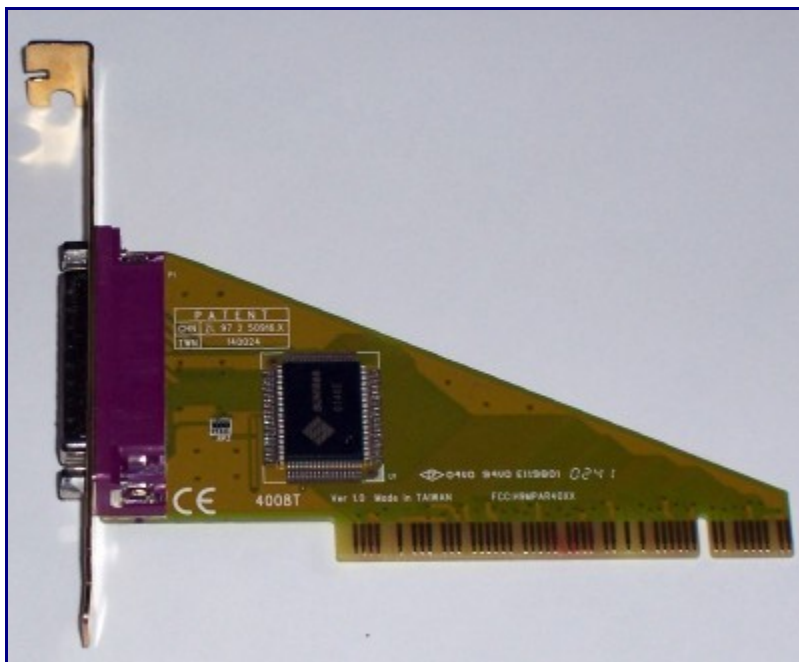
Disclaimer

La programmazione della porta parallela è un campo estremamente affascinante, ma a cui avvicinarsi con cautela. Il chip che gestisce la porta parallela è sulla scheda madre, e in molti casi gestisce anche altri componenti, quali dischi, interfacce di I/O ecc. Se non si ha abbastanza esperienza con il saldatore e si vuole collegare un dispositivo fatto in casa alla parallela, è meglio NON collegarlo direttamente alla porta parallela sulla scheda madre. Ci vuole poco a creare un corto circuito che può danneggiare fisicamente e in modo irreparabile il chip sulla scheda madre, che in genere non è sostituibile e costringe alla sostituzione fisica dell'intera scheda madre. L'avvertenza è ancora più forte se si collega il proprio marchingegno elettronico ad un portatile nuovo di zecca. Ci vuole poco per trasformare il portatile nuovo di zecca in un oggetto da discarica se tra l'interfaccia di I/O e il proprio circuito collegato si

viene a creare un corto circuito. Il mio consiglio è di interporre tra il proprio dispositivo e la porta parallela sulla scheda madre un buffer tri-state che possa proteggere la scheda madre stessa, magari un tri-state integrato come il pic 74LS44, che ha il seguente schema elettrico:



O, meglio ancora, si può inserire nel proprio slot ISA o PCI della scheda madre una scheda del genere, che costa una decina di euro:

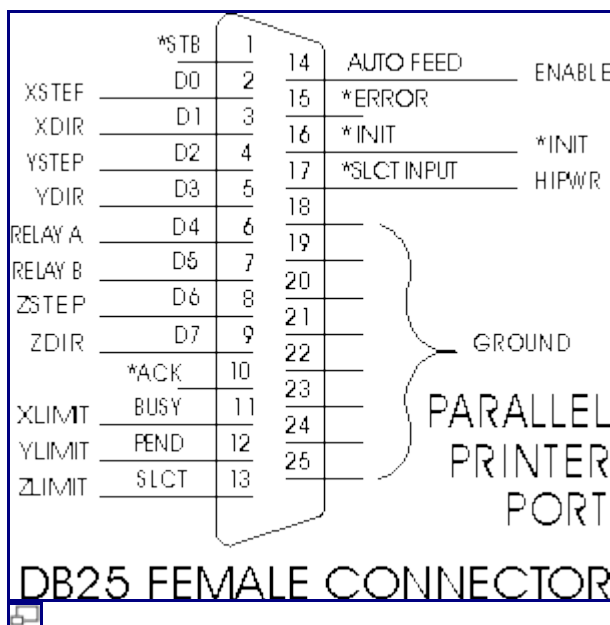


Se c'è qualcosa di sballato nel circuito salta la scheda PCI e con una decina di euro si può comprare una nuova, e almeno non salta la scheda madre.

Ancora, se si acquista un computer moderno è probabile che non sia presente l'interfaccia parallela. In questo caso si può rimediare con un adattatore USB-parallela come il seguente (una decina di euro):



Struttura della porta



Pin di una porta parallela

Su una porta parallela è possibile leggere o scrivere 1 **byte** (8 **bit**) per volta, informazione che viene salvata su un registro interno a 8 bit della porta. I pin che ci interessano in questa trattazione sono quindi quelli numerati da 2 a 9 (a ogni pin corrisponde un bit). I pin da 10 a 17 e 1 vengono usati come pin di controllo, per leggere lo status della porta o inviare segnali, mentre quelli da 18 a 25 sono tutti collegati a massa (tensione di riferimento nulla).

Individuazione dell'indirizzo della porta parallela

Ogni interfaccia di I/O di un calcolatore è mappata in memoria tramite un range di indirizzi, range che serve a identificare in modo non ambiguo il dispositivo e alla comunicazione del sistema con esso. La porta parallela è generalmente mappata agli indirizzi 0x378-0x37F. Per essere più sicuri, è meglio controllare. Su un sistema Unix

ciò è possibile dando un'occhiata al file `/proc/ioproports`. Per la porta parallela dovrebbe comparire una riga del genere:

```
0378-037f : lp1
```

Mentre invece su un sistema Windows si può controllare dalla gestione avanzata delle periferiche.

Primitive di sistema per la programmazione del dispositivo

Un sistema [Unix](#) mette a disposizione del programmatore una serie di primitive C per la gestione della porta parallela e non solo. Tramite le primitive che vedremo in questa sede sarà possibile programmare senza molti problemi qualsiasi dispositivo di I/O connesso alla propria macchina, ovviamente stando sempre attenti a quello che si sta facendo.

ioperm

Primitiva fondamentale per poter aprire un canale di comunicazione con la periferica di I/O è `ioperm()`, il cui compito è di settare o rimuovere i permessi di accesso ad una qualsiasi periferica di I/O. Come parametri prende

- L'indirizzo iniziale che identifica la periferica di I/O (0x378 nel caso della porta parallela)
- Il numero di byte assegnati alla periferica a partire dal byte iniziale (in genere per la parallela se ne considerano 4)
- Un intero che identifica se attivare o disattivare l'accesso alla periferica (1 per poter accedere alla periferica, 0 quando l'accesso non serve più)

C'è da ricordare che per utilizzare questa primitiva è necessario avere i privilegi di amministratore. Quindi l'applicazione che intende accedere alla periferica è necessario che sia di proprietà di root e abbia il bit UID settato, in modo da poter accedere alla periferica con `ioperm()`. Esempio di uso:

```
// Indirizzo di partenza della periferica
#define PORT 0x378

...

int uid=geteuid();

// Se non sono root, setto i privilegi di root
if (uid)
    setreuid(0,0);

// Accedo alla periferica
if (ioperm(PORT,3,1)==-1) {
```

```

    perror ("Errore nell'accesso alla periferica\n");
    exit(1);
}

// Torno a settare i permessi di utente normale
if (uid)
    setreuid(uid,uid);

```

inb o outb

Per leggere un byte per volta su una porta di I/O e scriverli il kernel mette a disposizione le primitive *inb* e *outb*, utilizzabili nel proprio codice C a patto di includere l'header `<asm/io.h>` (in quanto sono parallele alle istruzioni ASM *IN* e *OUT*). La loro sintassi è la seguente:

```

short int inb (int port);

void outb (short int val, int port);

```

inb legge un byte dalla porta all'indirizzo *port* (precedentemente aperta) e ritorna il valore letto. *outb* invece scrive il byte *val* sulla porta all'indirizzo *port*.

Per applicazioni diverse dalla porta parallela (che avendo 8 data pin può interagire con 1 byte per volta) è possibile leggere o scrivere sulla periferica una [word](#) per volta o una [double word](#) rispettivamente con le primitive *inw-outw* o *inl-outl*, che hanno la stessa sintassi di quelle già viste.

Esempio pratico

Prendiamo un esempio facile facile in esame. Immaginiamo di aver collegato alla porta parallela un [led](#), collegato in modo che sia polarizzato in diretta (terminale positivo sul data pin n.1 della porta parallela e terminale negativo collegato ad un qualsiasi ground pin della porta parallela). Vogliamo che il nostro led si accenda a intermittenza, diciamo pure con un intervallo di 1 secondo tra un cambiamento e l'altro (in pratica vogliamo sfruttare la porta parallela come un generatore di onde quadre). La cosa è possibilissima con le conoscenze che abbiamo finora. Ecco il codice in C:

```

#include <stdio.h>
#include <stdlib.h>
#include <asm/io.h>

// Indirizzo della parallela
#define PORT          0x378

main() {
    // Controllo che utente sono
    int uid=geteuid();

```

```

        // Se non sono root, acquisisco i privilegi con un
setreuid()
    if (uid)
        setreuid(0,0);

    // Attivo la porta
    if (ioperm(PORT,3,1)<0)
        exit(1);

    // Torno a essere utente normale
    setreuid(uid,uid);

    // Ciclo infinito
    while (1) {
        // Scrivo 0000 0001 sulla porta
        // in modo da alimentare solo il data pin n.1
        // dove è collegato il nostro diodo
        outb(0xFF,PORT);

        // Aspetto un secondo
        sleep(1);

        // Disattivo i data pin scrivendo 0000 0000 sulla
porta
        outb(0,PORT);

        // Aspetto un secondo
        sleep(1);
    }
}

```

Interfacciamento tra C e MySQL

.

È possibile interfacciarsi con un database MySQL tramite alcune funzioni C, messe a disposizione dagli sviluppatori dello stesso DBMS.

Applicazione pratica

Per vedere come è possibile interfacciarsi con un database MySQL tramite il C, prendiamo subito in esame un esempio pratico. Abbiamo un database MySQL chiamato "esami", che gestisce gli esami tenuti in una certa facoltà. Il database contiene queste tabelle:

---CORSO---

Field	Type	Null	Key	Default	Extra
codcorso	int(11)	NO	PRI	NULL	auto_increment
nomeC	char(40)	YES		NULL	
coddoc	int(11)	NO	MUL		

---DOCENTE---

Field	Type	Null	Key	Default	Extra
coddoc	int(11)	NO	PRI	NULL	auto_increment
nomeD	char(20)	YES		NULL	
cognomeD	char(20)	YES		NULL	

---ESAME---

Field	Type	Null	Key	Default	Extra
coddoc	int(11)	NO	PRI	0	
codcorso	int(11)	NO	PRI	0	
matr	int(11)	NO	PRI	0	
voto	int(11)	YES		NULL	

---STUDENTE---

Field	Type	Null	Key	Default	Extra
-------	------	------	-----	---------	-------

matr	int(11)	NO	PRI	NULL	auto_increment
nomeS	char(20)	YES		NULL	
cognomeS	char(20)	YES		NULL	
anno_corso	int(1)	YES		NULL	

La nostra applicazione dovrà interfacciarsi con questo database in modo da poter prelevare informazioni al suo interno. Innanzitutto, prima di effettuare qualsiasi operazione sul database, bisogna inizializzare il descrittore del database, usato all'interno dell'applicazione, tramite la funzione `mysql_init()` (definita, come tutte le funzioni che operano su database MySQL, in `mysql/mysql.h`), che ha questa sintassi:

*MYSQL *mysql_init(MYSQL *mysql)*

dove `MYSQL` è un tipo di dato primitivo usato dalla libreria MySQL e, in questo caso, rappresenta il descrittore del nostro database. La funzione ritorna `NULL` quando non è possibile creare il descrittore. Esempio di utilizzo:

```
#include <mysql/mysql.h>

.....

MYSQL db;

if (!mysql_init(&db)) {
    printf ("Errore nell'inizializzazione del database\n");
    exit(1);
}
```

A questo punto bisogna collegarsi fisicamente al database sull'host in questione, usando la funzione `mysql_real_connect`, che ha la seguente sintassi:

*MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd, const char *db, unsigned int port, const char *unix_socket, unsigned long client_flag)*

dove `*mysql` rappresenta l'indirizzo del descrittore del database che abbiamo inizializzato prima, `host` l'IP o il nome dell'host sul quale è ospitato il database, `user` l'username con cui accedere al database e `passwd` la sua password corrispondente, `db` l'eventuale database a cui collegarsi (se sulla macchina esistono più istanze di MySQL, altrimenti si può tranquillamente lasciare a `NULL`), `port` l'eventuale porta a cui collegarsi (se il database è in ascolto su una porta diversa da quella di default, altrimenti si può tranquillamente lasciare a 0), `unix_socket` l'indirizzo dell'eventuale socket da utilizzare per la connessione (in genere si può lasciare a `NULL`), `client_flag` un intero che rappresenta eventuali informazioni aggiuntive da passare al db (in genere si lascia a 0, per esigenze particolari sul sito developer di MySQL c'è una voce dedicata ai possibili valori che può assumere questo flag, nel caso di esigenze particolari). Nel nostro caso di esempio, ci collegheremo al server MySQL presente sul nostro host locale (`localhost`), sfruttando il descrittore db creato prima, lo username 'root' e la password 'prova':

```

char *db_host="localhost";
char *db_user="root";
char *db_pass="prova";

if (!mysql_real_connect(&db, db_host, db_user, db_pass, NULL, 0,
NULL, 0))
    printf ("Errore di connessione al database su %s\n",db_host);
else
    printf ("Connessione avvenuta con successo al database su
%s\n",db_host);

```

A questo punto selezioniamo il database da utilizzare sull'host a cui ci siamo collegati. Il nostro database era quello dedicato agli esami della facoltà, chiamato "esami". Per selezionare un database da un descrittore già aperto usiamo la funzione `mysql_select_db`:

```

if (mysql_select_db(&db,db_name))
    printf ("Errore di connessione al database %s\n",db_name);
else
    printf ("Connessione avvenuta con successo al database
%s\n",db_name);

```

Ora la connessione al database è avvenuta con successo, e il database è pronto ad accettare le nostre richieste. Per fare una query SQL al database si usa la funzione `mysql_real_query`, a cui bisogna passare i seguenti parametri:

- Indirizzo del descrittore del db
- Query SQL, sotto forma di stringa
- Lunghezza della query

Esempio: vogliamo interrogare il database in modo da ottenere il numero di matricola, il nome e il cognome di tutti gli studenti che hanno sostenuto almeno un esame, con il nome dell'esame superato e il voto corrispondente. Si tratta di fare un join tra 3 tabelle del db: `studente` (dal quale prelevo il nome, il cognome e la matricola degli studenti), `corso` (dal quale prelevo il nome del corso) e `esame` (dal quale prelevo il voto). Ovviamente la condizione di join è che il codice del corso di esame sia uguale a quello di corso, e la matricola dello studente sia uguale a quella di esame. Ordinando i risultati in modo crescente secondo il codice del corso, la query diventa così:

```

char *query =
    "select s.matr,s.nomeS,s.cognomeS,c.nomeC,e.voto "
    "from studente s,corso c,esame e "
    "where e.codcorso=c.codcorso "
    "and e.matr=s.matr "
    "order by c.codcorso";

```

Il codice per eseguirla diventa così:

```

if ( mysql_real_query (&db, query, (unsigned int) strlen(query)) )
{

```

```

    printf ("Errore nell'esecuzione della query %s\n",query);
    exit(2);
}

```

Ora è possibile salvare il risultato della query in una variabile apposita (di tipo predefinito MYSQL_RES), tramite la funzione mysql_store_result, quindi contare il numero di campi letti attraverso mysql_num_fields e salvare il nome di ogni campo (es. nomeS, cognomeS, voto...) in una variabile apposita (di tipo MYSQL_FIELD) attraverso la funzione mysql_fetch_fields. Ecco quindi come ottenere i nomi di tutti i campi letti dalla query all'interno del database e stamparli su schermo uno per uno (la formattazione del testo non sarà ottimale, ma è solo per capire come funziona il procedimento):

```

MYSQL_RES *res;
MYSQL_FIELD *f;
int i;

.....

res = mysql_store_result(&db);
f = mysql_fetch_fields(res);

for (i=0; i<mysql_num_fields(res); i++)
    printf ("%s\t",f[i].name);

```

Ora stampiamo i contenuti effettivi di ogni riga della query. Per fare ciò, usiamo un altro tipo di dato primitivo di MySQL (MYSQL_ROW) e usiamo la funzione mysql_fetch_row. Per leggere tutte le righe date in output dalla query il codice diventa quindi qualcosa del genere:

```

MYSQL_ROW row;

.....

// Finché ci sono righe da leggere...
while ((row=mysql_fetch_row(res))) {
    // ...per ogni riga letta...
    for (i=0; i<n; i++)
        //...stampare il contenuto i-esimo
        printf ("[%s]\t", row[i]);
    printf ("\n");
}

```

A questo punto il nostro interfacciamento con il database è completo, e ripuliamo sia il risultato della query sia l'identificatore della connessione con il database, attraverso le funzioni mysql_free_result e mysql_close:

```

mysql_free_result (res);
mysql_close(&db);

```


Ecco il codice completo dell'esempio:

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

main() {
    char *db_host="localhost";
    char *db_user="root";
    char *db_pass="prova";
    char *db_name="esami";
    char *query =
        "select s.matr,s.nomeS,s.cognomeS,c.nomeC,e.voto "
        "from studente s,corso c,esame e "
        "where e.codcorso=c.codcorso "
        "and e.matr=s.matr "
        "order by c.codcorso";

    int i;
    unsigned int n;

    MYSQL db;
    MYSQL_RES *res;
    MYSQL_ROW row;
    MYSQL_FIELD *f;

    if (mysql_init(&db)==NULL) {
        printf ("Errore nell'inizializzazione del
database\n");
        exit(1);
    }

    if (!mysql_real_connect(&db, db_host, db_user, db_pass,
NULL, 0, NULL, 0))
        printf ("Errore di connessione al database su
%s\n",db_host);
    else
        printf ("Connessione avvenuta con successo al
database su %s\n",db_host);

    if (mysql_select_db(&db,db_name))
        printf ("Errore di connessione al database
%s\n",db_name);
    else
        printf ("Connessione avvenuta con successo al
database %s\n",db_name);

    if ( mysql_real_query (&db, query, (unsigned int)
strlen(query)) ) {
        printf ("Errore nell'esecuzione della query
%s\n",query);
        exit(2);
    }
}
```

```

    res = mysql_store_result(&db);
    n = mysql_num_fields(res);
    f = mysql_fetch_fields(res);

    printf ("\n");

    for (i=0; i<n; i++)
        printf ("%s\t",f[i].name);

    printf ("\n");

    while ((row=mysql_fetch_row(res)) {
        for (i=0; i<n; i++)
            printf ("[%s]\t", row[i]);
        printf ("\n");
    }

    mysql_free_result (res);
    mysql_close(&db);
}

```

Ovviamente, le funzioni contenute in questo codice di esempio si possono riutilizzare per effettuare delle query su qualsiasi db, e anche per effettuare operazioni di creazione, inserimento e aggiornamento di record all'interno di un database. La documentazione completa per le funzioni di interfacciamento tra MySQL e C la potete trovare [qui](#).

CGI in C

Utilizzando il meccanismo dei *CGI* (**Common Gateway Interface**) è possibile innescare vere e proprie applicazioni che hanno la libertà di svolgere qualsiasi funzione eseguibile sul web server da un programma, per poi restituire un risultato in forma di pagina [HTML](#). Il C consente di fare operazioni del genere, in modo forse meno avanzato rispetto a linguaggi dedicati come PHP o Perl ma estremamente flessibile, date le sue caratteristiche.

Pagine statiche e pagine dinamiche

Sono dette pagine statiche quelle pagine presenti sul web server che non richiedono alcuna elaborazione da parte del browser se non quella di prendere la pagina così com'è e inviarla al browser. Generalmente queste pagine sono scritte in linguaggi detti di "markup", gli esempi più palesi sono [HTML](#) e [XML](#). Per pagina dinamica si intende una pagina non presente fisicamente sul disco rigido del Web Server, ma costruita al volo, per mezzo di un'applicazione (interfaccia CGI) o uno script dedicato (in PHP o ASP). Il meccanismo dei CGI estende e generalizza l'interazione request/response, cuore del protocollo HTTP. Ora descriviamo passo passo il meccanismo dei CGI, il processo si può dividere in 4 fasi:

1. *Invio della request* - Il browser (client HTTP) effettua una request a un server HTTP identificato dal seguente indirizzo o URL:

<http://www.nomesito.it/cgi-bin/hello.cgi?>

Possiamo identificare il server HTTP:<http://www.nomesito.it> e il riferimento alla procedura CGI:

cgi-bin/hello.cgi

La directory /cgi-bin è una sottodirectory della directory del web server che contiene le applicazioni CGI.

2. *Attivazione del CGI* - Il server HTTP (es. Apache, Netscape Server o Microsoft IIS) riceve la URL, la interpreta e lancia il processo (o thread) che esegue il CGI.

3. *Risposta del CGI* - Il risultato della computazione deve dar luogo a una pagina HTML di risposta, che il CGI invia verso il suo Standard Out (per i CGI lo STDOUT viene intercettato dal server HTTP) tenendo conto di quale deve essere il formato di una response HTTP.

4. *Risposta del server HTTP* - Sarà poi il server HTTP ad inviare la response verso il client che aveva effettuato la request.

Passiamo adesso a vedere come si scrive un CGI in C, prendendo come spunto un'applicazione che stampa all'interno di una pagina web 'hello world':

```
//Il CGI hello.c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Content-type: text/html\n\n"); /*informazione necessaria
per la response*/

    /*Inviemo su STDOUT i tag HTML*/
    printf("<html>\n"
           "<head>\n"
           "<title>Hello World!</title>\n"
           "</head>\n"
           "<body>\n"
           "<h1><p align=\"center\">Hello World</p></h1>\n"
           "</body>\n"
           "</html>\n");

    return 0;
}
```

Compilando questo programma all'interno della directory /cgi-bin del nostro server web

```
gcc -o hello.cgi hello.c
```

otteniamo un eseguibile CGI che possiamo richiamare all'interno del nostro browser nel modo visto sopra

<http://www.miosito.it/cgi-bin/hello.cgi>

L'esame del codice non è nulla di assurdo, tenendo sempre presente che lo STDOUT di una CGI viene rediretto direttamente al client HTTP. Degna di nota è questa riga:

```
printf("Content-type: text/html\n\n");
```

Il suo scopo è quello di specificare al client HTTP il tipo di contenuto che si sta per inviare (in questo caso del testo HTML). Senza questa specifica il client non sa come comportarsi e stamperà una pagina bianca.

Si notino le due linee vuote; sono assolutamente necessarie, per le convenzioni del protocollo HTTP, ma sono spesso dimenticate. Il segnale di STDOUT viene quindi intercettato dal browser, e viene generata una pagina web con i contenuti specificati.

Prendiamo ora in esame un rudimentale orologio che invia al client HTTP una pagina web contenente l'ora del server sfruttando [la libreria time.h](#):

```

/* Ora quasi esatta */
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[]) {
    time_t bintime;
    struct tm *curtime;

    printf("Content-type: text/html\n\n");

    printf("<html>\n");
    printf("<head>\n");
    printf("<title>Orologio</title>\n");
    printf("</head>\n");
    printf("<body>\n");
    printf("<h1>\n");
    time(&bintime);
    curtime = localtime(&bintime);
    printf("Data e ora: %s\n", asctime(curtime));
    printf("</h1>\n");
    printf("</body>\n");
    printf("</html>\n");

    return 0;
}

```

Richieste GET e POST

L'utilità principale delle CGI, e dei linguaggi web-oriented come PHP e ASP, è quella di poter anche interagire con l'utente finale, leggendo dati inseriti da un utente via form o tramite altre vie. Per far questo il protocollo HTTP prevede due strade: il metodo *GET* e il metodo *POST*.

GET

Nel metodo *GET* i dati inseriti dall'utente o previsti dal programmatore vengono caricati nell'URL, e il loro contenuto, a livello del sistema server, finisce in una variabile d'ambiente chiamata *QUERY_STRING*. Immaginiamo ad esempio di avere il seguente codice HTML (ricordate che la scelta del metodo, GET o POST, va fatta a livello del codice del form HTML):

```

<form method=GET action=/cgi-bin/cgil>
  Come ti chiami? <input type="text" name="nome"><br>
  <input type="submit" value="Clicca">
</form>

```

È un semplice form HTML che chiede all'utente di turno come si chiama e invia la stringa inserita dall'utente all'eseguibile 'cgi1' tramite metodo GET (per convenzione gli eseguibili CGI si mettono nella directory /cgi-bin del server). Se salviamo questa

pagina come 'user.html' dopo aver cliccato sul tasto 'Clicca' la richiesta verrà inoltrata tramite metodo GET a cgi1, che quindi verrà richiamato nel seguente modo:

http://www.miosito.org/cgi-bin/my_cgi?nome=Nome_inserito_dall_utente

Nel caso ci fossero stati più campi oltre al nome (ad esempio un campo 'password') avremmo avuto una cosa del genere:

http://www.miosito.org/cgi-bin/cgi1?nome=Nome_inserito_dall_utente&password=Password_inserita

In pratica quando inviamo una richiesta tramite GET l'eseguibile CGI che viene richiamato (o lo script PHP/ASP) viene richiamato passando nell'URL una struttura del genere:

http://www.miosito.org/cgi-bin/my_cgi?campo1=val1&campo2=val2&campo3=val3.....

Ora immaginiamo che il nostro eseguibile cgi1 debba leggere il nome inserire dall'utente e generare per lui una pagina HTML di benvenuto (es. 'Benvenuto pippo!'). Ecco un potenziale codice C come potrebbe essere:

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

// Funzione che converte eventuali caratteri speciali
// all'interno della stringa inserita dall'utente in
// caratteri ASCII leggibili
// Prende come parametri la stringa sorgente, la stringa
// di destinazione e la lunghezza della stringa da 'uncodare'
void unencode (char *src, char *dest, int len);

// Funzione per il prelevamento di
// un campo da una query
// Prende come parametri la query in cui cercare
// e il nome del campo da cercare (in questo caso 'nome')
char* get_field(char *query, char *field);

main() {
    char *query,*nome;
    int len;

    // Genero la pagina HTML
    printf ("Content-type: text/html\n\n");
    printf ("<html>\n"
           "<head>\n"
           "<title>Pagina di benvenuto</title>\n"
           "</head>\n"
           "<body>\n");

    // Se la richiesta GET non contiene niente, la pagina è stata
    richiamata
```

```

// in modo errato, quindi esco
if ((query=getenv("QUERY_STRING"))==NULL) {
    printf ("<h3>Pagina richiamata in modo errato</h3>\n"
           "</body></html>\n");
    exit(1);
}

// Controllo la lunghezza della query e
// genero una stringa lunga quanto la query
// che conterrà il nome inserito dall'utente

// Ricordiamo che query ora sarà una stringa
// del tipo 'nome=pippo'
len=strlen(query);
nome = (char*) malloc(len*sizeof(char));

// Ora nome conterrà il campo 'nome' della query
nome=get_field (query,"nome");

printf ("<h3>Benvenuto %s!</h3>\n"
        "</body></html>\n",nome);

exit(0);
}

char* get_field(char *query, char *field) {
    int i,j,len,pos;
    char *tmp,*input;

    // len è pari alla lunghezza della query+1
    len = strlen(query)+1;

    // tmp sarà il pattern di ricerca all'interno della query
    // Nel nostro caso andrà a contenere la stringa 'nome='
    tmp = (char*) malloc( (strlen(field)+1)*sizeof(char) );

    // input è lunga quanto la query, e andrà a contenere
    // il campo da noi ricercato
    input = (char*) malloc(len*sizeof(char));

    // tmp <- nome=pippo
    sprintf (tmp, "%s=", field);

    // Se all'interno della query non c'è il campo richiesto, esco
    if (strstr(query,tmp)==NULL)
        return NULL;

    // Cerco la posizione all'interno della query
    // in cui è stato trovato il campo nome
    pos = ( (int) strstr(query,tmp) - (int) query) + (strlen(field)
+1);

    // Controllo quanto è lungo il pattern nome=blablabla

```

```

        // Questo ciclo termina quando viene incontrato un '&'
all'interno
        // della query (ovvero quando comincia un nuovo campo) o quando
la stringa è terminata
        // Alla fine i conterrà il numero di caratteri totali nel
pattern di ricerca
        for (i=pos; ; i++) {
            if (query[i]=='\0' || query[i]=='&') break;
        }

        // Salvo il contenuto della query che mi interessa in input
        for (j=pos; j<i; j++)
            input[j-pos]=query[j];

        // 'unencodo' input, rendendo eventuali caratteri speciali
umanamente leggibili
        unencode(input,input,len);

        // Ritorno input
        return input;
    }

void unencode(char *src, char *dest, int len) {
    int i,code;

    // Ciclo finché non ho letto tutti i caratteri specificati
    for (i=0; i<len; i++, src++, dest++) {
        // Se il carattere corrente di src è un '+', lo converto
in uno spazio ' '
        if (*src=='+') *dest=' ';

        // Se il carattere corrente è un '%'
        else if (*src=='%') {
            // Se il carattere successivo non è un carattere
valido,
            // il carattere di destinazione sarà un '?',
            // altrimenti sarà il carattere ASCII corrispondente
            if (sscanf(src+1, "%2x", &code) != 1) code='?';
            *dest = (char) code;

            // Leggo il prossimo carattere
            src += 2;
        }

        // Se è un carattere alfanumerico standard e non un
carattere speciale,
        // allora il carattere di destinazione è uguale a quello
sorgente
        else *dest=*src;
    }

    // Termino la stringa di destinazione
    dest[len]='\0';
}

```


}

La funzione `unencode` è indispensabile. Infatti, se l'utente dovesse inserire degli spazi o dei caratteri speciali qualsiasi all'interno del form (ovvero caratteri non alfanumerici) questi all'interno della `QUERY_STRING` verranno tradotti con i codici ASCII corrispondenti preceduti da un '%'. Ad esempio, se l'utente dovesse inserire 'pippo pappo', la query diventera 'nome=pippo+pappo'. Per convertire il carattere a quello inizialmente inserito dall'utente è quindi necessario passare per `unencode`.

Per comodità conviene tenersi le funzioni `get_field` e `unencode` da qualche parte pronte per l'uso, vista la loro utilità all'interno delle CGI in C. Personalmente ho sviluppato una piccola libreria (`cgic`) che contiene tutte queste funzioni utili al programmatore di CGI in C, senza che ci sia bisogno di reinventare la ruota ogni volta. Il link al pacchetto lo potete trovare alla fine di questo articolo.

POST

La scelta tra metodo GET e metodo POST è legata ad un preciso criterio di programmazione, che prevede che un GET venga scelto se e soltanto se i campi all'interno del form sono idempotenti tra di loro. Questa è la regola formale. La regola empirica insegna che le richieste GET vanno usate solo per campi di piccole dimensioni (ad esempio, form con checkbox, con variabili contenenti i nomi di pagine esterne da richiamare all'interno del codice, con campi contenenti piccole stringhe e così via). Non è una buona idea utilizzare richieste GET, ad esempio, per inviare un messaggio postato da un utente in un forum, in quanto verrà fuori un URL lunghissimo senza senso. È anche rischioso usare GET per form di login, in quanto i dati di autenticazione passerebbero in chiaro nell'URL. In tutti questi casi (e altri) è consigliabile l'uso del metodo POST.

Il metodo POST genera una query string che è uguale in tutto e per tutto a quella generata dal metodo GET (nel nostro caso, sempre `nome=nome_inserito`). La differenza è il metodo GET prevede che la query venga inviata al server tramite la variabile d'ambiente `QUERY_STRING`, mentre a livello client viene integrata nell'URL stesso. Il metodo POST invece prevede che la query venga inviata dal client al server all'interno del pacchetto HTTP stesso, e viene letta dal server come se fosse un normale input (quindi con `scanf`, `gets` o `fgets`). Prima di inviare la query vera e propria il client invia al server una stringa che identifica la lunghezza della query che sta per essere inviata. Questa stringa viene salvata dal server nella variabile d'ambiente `CONTENT_LENGTH`. In questo modo il server riceve la lunghezza della query che sta per essere inviata, prepara un buffer di dimensioni adatte e quindi legge la query con le funzioni per la lettura dell'input già viste. Dopo la procedura rimane uguale (ovvero lettura del contenuto di una variabile con un metodo come `get_field` e decodifica dei caratteri con un metodo come `unencode`).

Ecco un esempio di codice HTML che invia i dati di un form tramite metodo POST (esempio tipico, l'invio di un messaggio in un form che viene poi inviato ad un

eseguibile CGI e stampato su schermo):

```
<form method="POST" action="/cgi-bin/cgi2">
  Inserisci qui il tuo messaggio:<br>
  <textarea cols=50 rows=4 wrap="physical" name="msg"/><br>
  <input type="submit" value="Invia"/>
</form>
```

Ed ecco una potenziale applicazione CGI per elaborarlo:

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

// Il contenuto delle funzioni get_field() e unencode()
// è lo stesso visto nel codice precedente
void unencode (char *src, char *dest, int len);
char* get_field(char *query, char *field);

main() {
  char *query,*msg;
  int len;

  // Genero la pagina HTML
  printf ("Content-type: text/html\n\n"
         "<html>\n"
         "<head>\n"
         "<title>Messaggio inserito</title>\n"
         "</head>\n"
         "<body>\n");

  // Se la variabile d'ambiente CONTENT_LENGTH è nulla, oppure
  // la conversione in intero con sscanf non produce un intero
  // valido, esco
  if (getenv("CONTENT_LENGTH") == NULL || sscanf ( (char*)
getenv("CONTENT_LENGTH"), "%d", &len) != 1) {
    printf ("Contenuto non valido\n"
           "</body></html>\n");
    exit(1);
  }

  // Query sarà grande tanto quanto il pacchetto che sta per
  // inviarmi il client
  query = (char*) malloc (++len*sizeof(char));

  // Leggo la richiesta POST inviata dal client
  // come se fosse un normale input con fgets
  fgets (query,len,stdin);

  // Leggo il campo
  msg = get_field (query,"msg");

  printf ("Messaggio inserito:<br>\n%s\n",msg);
  exit(0);
}
```

}

Link esterni

È possibile usare librerie già pronte per l'uso di eseguibili CGI in C (come `get_field`, `unencode` e altre), senza dover reinventare la ruota e riscrivere funzioni da zero di volta in volta. [Qui](#) trovate la mia libreria, testata con successo su sistemi Unix, che già contiene molte funzioni comode per la scrittura di eseguibili CGI.

Catturare pacchetti con le librerie PCAP

.

La libreria *PCAP*, disponibile per sistemi Unix e Microsoft e scaricabile gratuitamente da [qui](#), offre al programmatore l'opportunità di usare all'interno del suo codice funzioni per la cattura e la gestione di tutti i pacchetti che transitano su un'interfaccia di rete. Su questa libreria di basa il celeberrimo *tcpdump*, il più classico degli sniffer/monitor di rete (e infatti la libreria è messa a punto dagli stessi sviluppatori di *tcpdump*), e anche sniffer più avanzati come *Ethereal/Wireshark*, e un porting in [Java](#) chiamato *Jpcap* che offre le funzionalità della libreria stessa anche nell'ambiente del linguaggio Sun.

Compilare e linkare programmi con le librerie PCAP

La procedura che vedremo qui mostra come compilare programmi con le librerie PCAP in ambiente Unix e con gcc, ma in generale è valida per qualsiasi libreria esterna installata sul sistema. Una volta installata la libreria è necessario includere nei sorgenti che ne fanno uso l'header *pcap.h*, e quindi compilare e linkare il programma con l'opzione *-lpcap* passata a gcc:

```
gcc -o nome_programma sorgente.c -lpcap
```

Inoltre i programmi che fanno uso delle librerie PCAP, accedendo alle interfacce di rete con i privilegi di superutente, hanno bisogno di essere avviati con i privilegi di root su sistemi Unix, di amministratore su sistemi Windows.

Trovare un'interfaccia di rete

La prima informazione che bisogna passare alle funzioni di PCAP è l'interfaccia di rete su cui si intende effettuare lo sniffing o il monitoring. Ciò è possibile con la funzione *pcap_lookupdev*, che trova automaticamente la prima interfaccia di rete disponibile sul sistema:

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

```
main(int argc, char **argv) {
```

```

char *dev, errbuf[PCAP_ERRBUF_SIZE];

dev = pcap_lookupdev(errbuf);

if (!dev) {
    printf ("Errore: nessuna interfaccia di rete trovata sul
sistema: %s\n",errbuf);
    exit(1);
}

printf ("Dispositivo di rete %s disponibile sul sistema\n",dev);
}

```

La funzione `pcap_lookupdev` in pratica cerca il miglior dispositivo di rete disponibile sul sistema e ritorna una stringa ad esso associata, altrimenti NULL se non c'è nessun dispositivo di rete. La stringa `errbuf`, di lunghezza `PCAP_ERRBUF_SIZE` definita in `pcap.h`, serve a contenere eventuali messaggi di errore.

Per trovare invece tutte le interfacce di rete sul sistema possiamo usare la funzione `pcap_findalldevs`, che prende come parametri un puntatore a puntatore a un tipo di dato `pcap_if_t` e il solito buffer di errore. `pcap_if_t` non è altro che un tipo di dato che identifica un'istanza della struttura `pcap_if` così definita:

```

struct pcap_if {
    struct pcap_if *next;
    char *name; /* name to hand to "pcap_open_live()" */
    char *description; /* textual description of interface, or
NULL */
    struct pcap_addr *addresses;
    bpf_u_int32 flags; /* PCAP_IF_ interface flags */
};

```

Ecco quindi un codice per visualizzare tutte le interfacce di rete disponibili su un sistema:

```

pcap_if_t *ifc;
char errbuf[PCAP_ERRBUF_SIZE];
struct sockaddr_in *addr;

pcap_findalldevs (&ifc,errbuf);

printf ("Interfacce di rete trovate sul sistema:\n\n");

// Finché ci sono interfacce da visualizzare...
while (ifc->next) {
    // ...stampo nome e descrizione
    printf ("%s: %s\n",ifc->name,ifc->description);

    // Finché ci sono indirizzi associati all'interfaccia di rete...
    while (ifc->addresses) {
        // ...stampo gli indirizzi
        addr = (struct sockaddr_in*) ifc->addresses->addr;
    }
}

```

```

printf ("Indirizzo: %s\n",inet_ntoa(addr->sin_addr.s_addr));

// Passo all'indirizzo successivo
ifc->addresses=ifc->addresses->next;
}

// Passo all'interfaccia di rete successiva
ifc=ifc->next;
}

```

Per verificare l'indirizzo e la netmask associate ad un'interfaccia di rete conviene usare la funzione *pcap_lookupnet()*, che prende come argomenti

- Il nome dell'interfaccia di rete
- Un puntatore ad una variabile a 32 bit che identifica la rete
- Un puntatore ad una variabile a 32 bit che identifica la netmask
- errbuf

La funzione ritorna -1 nel caso non ci sia nessun indirizzo associato ad un'interfaccia di rete. Esempio, per trovare l'indirizzo associato all'interfaccia di rete eth0:

```

bpf_u_int32 net,mask;
char errbuf[PCAP_ERRBUF_SIZE];

.....

if (pcap_lookupnet("eth0",&net,&mask,errbuf)==-1) {
    printf ("Nessun indirizzo associato a eth0: %s\n",errbuf);
    exit(1);
}

```

Sniffing

La funzione messa a disposizione dalle PCAP per l'apertura di un dispositivo di rete per lo sniffing è *pcap_open_live()*. Tale funzione prende come argomenti:

- Il nome del dispositivo di rete su cui effettuare lo sniffing
- Il numero massimo di byte da catturare per ogni sessione
- Un valore booleano (*promisc*) che se settato a 1 pone il dispositivo di rete in modalità promiscua. Se lasciato a 0 di default PCAP snifferà solo il traffico di rete diretto verso la propria interfaccia di rete
- *to_ms*, che identifica il numero di secondi passati i quali la sessione di sniffing va in timeout. Se settato a 0 non ci sarà nessun timeout per la sessione di sniffing
- errbuf

La funzione ritorna un puntatore ad una variabile di tipo *pcap_t*, che per il resto del listato sarà il descriptor della nostra sessione di sniffing, oppure NULL in caso di errore. Esempio pratico:

```

pcap_t *sniff;
char errbuf[PCAP_ERRBUF_SIZE];

.....

if (!(sniff=pcap_open_live("eth0",1024,1,0,errbuf)) {
    printf ("Errore nella creazione di una sessione di sniffing su
eth0: %s\n",errbuf);
    exit(1);
}

printf ("Sessione di sniffing creata con successo\n");

```

Questo codice apre una sessione di sniffing sul dispositivo eth0 in modalità promiscua, leggendo 1024 byte per volta, senza un timeout impostato per la sessione e con un eventuale buffer di errore salvato in *errbuf*. Se l'esecuzione del codice va a buon fine in *sniff* troveremo un descrittore per la nostra sessione di sniffing da usare in seguito nel codice.

A questo punto è necessario *compilare* la sessione di sniffing specificando un eventuale filtro. Il filtro servirà nel caso in cui non vogliamo sniffare tutto il traffico di rete ma solo quello diretto o proveniente da una determinata porta, solo il traffico [TCP](#) o solo quello [UDP](#) e così via. Per compilare la sessione useremo la funzione *pcap_compile()* che prende i seguenti argomenti:

- Il descrittore della sessione di sniffing inizializzato precedentemente con *pcap_open_live()*
- Un puntatore ad una variabile di tipo *bpf_program*, dove verrà memorizzata la versione compilata della nostra sessione
- Una stringa di filtro
- La variabile booleana *optimize* che stabilisce se il filtro andrà ottimizzato o meno
- La netmask sulla quale verrà applicato il filtro (precedentemente inizializzata tramite *pcap_lookupnet()*)

La stringa di filtro sarà una stringa che identificherà il tipo di traffico da filtrare. La sintassi dettagliata è illustrata [qui](#). In generale, una stringa di filtro è strutturata nel seguente modo per il filtraggio su una determinata porta o protocollo:

```
[proto] [src|dst] [port numero_porta]
```

Ad esempio

```
tcp dst port 80
```

catturerà tutti e soli i pacchetti destinati alla porta 80 e scarcerà gli altri. Per il filtraggio sugli host la stringa di filtro sarà così costruita:

```
[host] [src|dst indirizzo_host]
```

Ad esempio

```
host dst 192.168.1.1
```

catturerà tutti e soli i pacchetti destinati all'host 192.168.1.1.

Nel caso non si voglia utilizzare un filtro e si vogliono sniffare tutti i pacchetti basterà settare la *filter_string* a NULL. Esempio pratico:

```
pcap_t *sniff;
bpf_u_int32 net,mask;
struct bpf_program filter;

// Questo per sniffare senza filtri
char *filter_string=NULL;

// Questo per filtrare, per esempio, solo il traffico destinato alla
// porta 80
char filter_string[] = "tcp dst port 80";

.....

pcap_compile (sniff,&filter,filter_string,0,net);
```

Quest'uso di *pcap_compile()* compilerà la nostra sessione di sniffing puntata da *sniff*, salverà la versione compilata su *filter* usando la stringa di filtro *filter_string*, senza opzioni di ottimizzazione e usando la netmask salvata in *net*.

Una volta creato e compilato il filtro è il caso di associarlo alla nostra sessione di sniffing. Questo si fa con la funzione *pcap_setfilter()*, che prende come argomenti

- Il descrittore di tipo *pcap_t* della sessione
- Il puntatore all'istanza di *bpf_program* nella quale è salvato il filtro appena compilato

```
pcap_setfilter (sniff,&filter);
```

Questa riga assocerà il descrittore della sessione creato in precedenza al filtro appena creato. A questo punto tutto è pronto per cominciare il ciclo di sniffing vero e proprio attraverso la funzione *pcap_loop()*, che prende come argomenti

- Il descrittore di tipo *pcap_t* della sessione
- Il numero di pacchetti da sniffare prima di uscire (0 per non imporre nessun limite)
- Il nome della funzione da richiamare quando giunge un pacchetto (la funzione che compierà le operazioni richieste su quel pacchetto)
- Eventuali argomenti aggiuntivi (generalmente settati a NULL)

Nel nostro caso:

```
pcap_loop (sniff,0,pack_handle,NULL);
```


dirà al compilatore di creare un ciclo di sniffing associato al descrittore *sniff*, senza imporre un limite massimo di pacchetti sniffati. Ogni volta che un pacchetto transita sull'interfaccia di rete viene richiamata la funzione *pack_handle()* per gestirla, funzione così definita:

```
void pack_handle (u_char *args, const struct pcap_pkthdr *p_info,
const u_char *packet) {
```

Questa è la sintassi standard di una funzione passata come argomento a *pcap_loop()*. Il primo argomento punta all'ultimo argomento specificato in *pcap_loop()*, ovvero agli eventuali argomenti aggiuntivi aggiunti (generalmente NULL). Il secondo argomento è un puntatore alla struttura *pcap_pkthdr*, che contiene informazioni circa il pacchetto appena sniffato. Questa struttura è così definita:

```
struct pcap_pkthdr {
    struct timeval ts; /* time stamp */
    bpf_u_int32 caplen; /* length of portion present */
    bpf_u_int32 len; /* length this packet (off wire) */
};
```

I campi disponibili sono

- Ora di cattura del pacchetto
- Lunghezza della porzione catturata
- Lunghezza totale del pacchetto

L'ultimo argomento è un buffer contenente il contenuto vero e proprio del pacchetto. In questo caso possiamo semplicemente scrivere all'interno della nostra funzione un `printf ("%s\n", packet);`

Un programma costruito in questo modo, con una tale funzione, farà un dump di tutti i pacchetti transitanti su un'interfaccia di rete su stdout. Possiamo fare qualcosa di più elaborato conoscendo gli standard dei pacchetti TCP/IP. Ad esempio nel caso di un'interfaccia ethernet risalire al MAC mittente e al MAC destinatario del pacchetto, tenendo presente che queste informazioni occupano i primi 12 byte del pacchetto, è un gioco da ragazzi:

```
printf ("MAC sorgente: %.2x:%2x:%.2x:%.2x:%.2x:%.2x\n",
packet[0], packet[1], packet[2], packet[3], packet[4], packet[5]);
```

```
printf ("MAC destinatario: %.2x:%2x:%.2x:%.2x:%.2x:%.2x\n",
packet[6], packet[7], packet[8], packet[9], packet[10], packet[11]);
```

Per maggiori informazioni sulla struttura dei pacchetti [TCP/IP](#) rimando alle sezioni apposite nell'area reti.

Packet injection

Tramite le PCAP è anche possibile fare packet injection, ovvero inserire su

un'interfaccia di rete pacchetti costruiti arbitrariamente. La funzione da usare è in questo caso *pcap_inject()*, che prende i seguenti argomenti:

- Il descrittore della sessione di sniffing
- Il buffer contenente il pacchetto costruito arbitrariamente
- La lunghezza del pacchetto

Si può quindi sniffare un pacchetto su un'interfaccia di rete, modificare il MAC o l'IP mittente e inviare una risposta al destinatario, che crederà che quel pacchetto venga dal mittente specificato. Tecnica ancora più efficace se abbinata a tecniche di ARP poisoning.

Introduzione ai sistemi fuzzy e alle reti neurali

.

Prerequisiti matematici

Dati due vettori di grandezze $X = x_1, \dots, x_n$ e $V = v_1, \dots, v_n$, tale che ogni elemento v_i è dipendente funzionalmente dal corrispondente elemento $x_i \forall i$, si dice *media pesata* il prodotto scalare di X per V :

$$\langle X, V \rangle = \sum_{i=1}^n x_i v_i$$

Sistemi fuzzy

I sistemi fuzzy sono sistemi che si ispirano alla logica fuzzy, una logica polivalente che si può considerare un ampliamento della logica booleana classica, che prende in esame non solo un numero discreto possibile di valori, come lo 0 e 1 nell'algebra di Boole, ma anche possibili valori "intermedi" non numerabili. La logica fuzzy si pone quindi come valida alternativa alla logica tradizionale nell'esame dei problemi reali, in cui i valori che possono assumere le variabili in gioco non sono numerabili, o almeno non facilmente numerabili.

Introduzione alle reti neurali

Le reti neurali sono un'applicazione dell'intelligenza artificiale relativamente vecchia (sono state teorizzate negli anni '60, per poi essere abbandonate sui primi anni '70 in seguito alla pubblicazione, da parte di M.Minsky e S.Papert, di *Perceptrons*, un libro che metteva in risalto le carenze della tecnologia), ma che ultimamente sta vivendo un periodo di rinascita in seguito al rinnovato interesse nei confronti delle tecniche di intelligenza artificiale e al perfezionamento di questa tecnologia stessa.

Il meccanismo delle reti neurali si ispira dichiaratamente a quello del sistema nervoso degli animali. Le reti neurali non sono progettate per "nascere imparate", né tantomeno per avere una grande precisione in fatto di calcolo (d'altronde la potenza di calcolo di un cervello umano verrebbe facilmente ridicolizzata da qualsiasi calcolatrice), ma sono progettate per apprendere. La fase di apprendimento di una

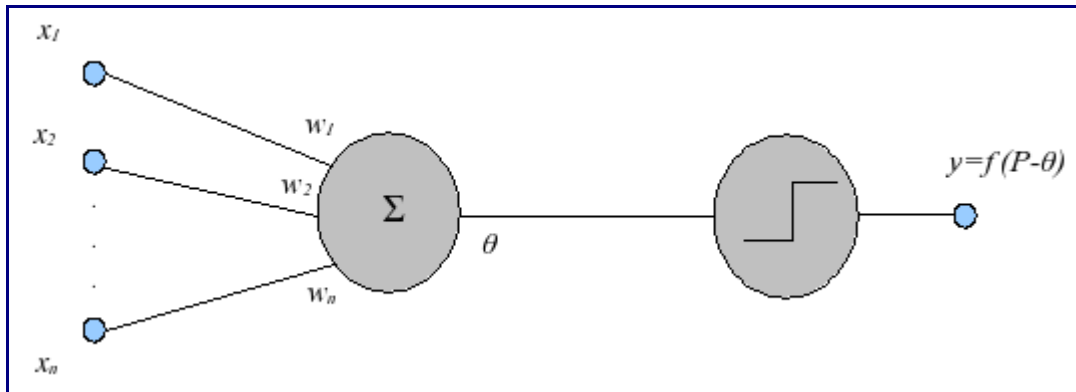
rete neurale si basa su un campione di dati (“training set”) che viene presentato alla rete stessa, spesso con i risultati che si desiderano ottenere. Ad esempio, se voglio addestrare una rete neurale a risolvere le 4 operazioni fondamentali, posso presentare in input alla rete diversi numeri, e poi i risultati che desidero ottenere con quei numeri. Sarà la rete stessa a imparare, gradualmente, i meccanismi che sono alla base dell'operazione che deve compiere. Ovviamente, più corposo sarà il training set della rete, più precisi saranno i risultati che si potranno ottenere una volta completato l'addestramento.

Le reti neurali, come accennavo prima, non sono molto usate nel campo del calcolo matematico-scientifico, proprio in base alla loro scarsa precisione da sistemi fuzzy, ma si rivelano utilissime (proprio in virtù delle loro caratteristiche fuzzy) nella risoluzione di problemi reali. Un cervello umano non saprà risolvere un integrale definito con il metodo dei rettangoli con la stessa rapidità con cui lo risolverebbe un calcolatore elettronico, ma può riconoscere con una facilità disarmante un cane da un albero, o la voce di un amico da lontano, anche se disturbata da altri rumori. Delle applicazioni simili in campo tecnologico le hanno anche le reti neurali, utili, ad esempio, per il riconoscimento visivo elettronico, per il riconoscimento vocale, e così via.

Struttura di una rete neurale

La struttura di una rete neurale si rifà esplicitamente a quella di una rete neurale umana. Nell'uomo i neuroni sono costituiti da un corpo cellulare (soma) e da dendriti che mettono il neurone in comunicazione con altri neuroni. In presenza di determinati segnali queste comunicazioni si attivano (sinapsi), con il rilascio di sostanze di tipo chimico-ormonale (neurotrasmettitori) che trasmettono lo stimolo da un neurone all'altro. L'input di un neurone non è altro che la media pesata di tutti i segnali provenienti in input dagli altri neuroni per il “peso sinattico” della sinapsi in questione, ovvero l’“importanza” che riveste quella sinapsi nel collegamento. In base a questo valore, chiamato potenziale post-sinattico, il neurone può rispondere con un valore di output, che può essere minore o maggiore in fatto di intensità rispetto al precedente (effetto “calmante” o “eccitante”) in base ai valori degli input in quel dato momento.

Un neurone artificiale ha una struttura simile:



Dove x_1, \dots, x_n sono gli input presentati al neurone o alla rete neurale, w_1, \dots, w_n sono i pesi sinattici delle singole connessioni (ovvero quanto quella connessione influenza il risultato finale). La media pesata degli input per i pesi sinattici delle singole connessioni fornisce il potenziale post-sinattico del neurone:

$$P = \sum_{i=1}^n w_i x_i$$

L'output y del neurone è dato da $f(P - \theta)$, dove θ è una soglia caratteristica del neurone, mentre f è una funzione di trasferimento. Le principali funzioni di trasferimento utilizzate nelle reti neurali sono la funzione a gradino, la funzione sigmoideale e la tangente iperbolica, tutte funzioni aventi codominio nell'intervallo $[0,1]$ (o $[-1,1]$ nel caso della tangente iperbolica).

La funzione a gradino, il tipo di funzione di trasferimento più semplice usata nelle reti neurali, è una funzione così definita:

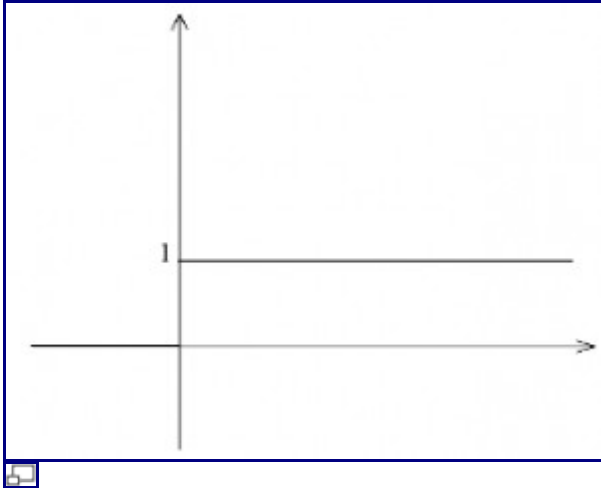


Grafico della funzione a gradino

$$f(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ 0 & \text{se } x < 0 \end{cases}$$

Usando questa funzione, il neurone emette un segnale $y=1$ quando $x = (P-\theta) \geq 0$, quindi $P \geq \theta$, mentre emette un segnale $y=0$ (quindi rimane inattivo) quando $P < \theta$.

Un'altra caratteristica funzione di trasferimento è la sigmoide, o curva logistica, di equazione

$$f(x) = \frac{1}{1 + e^{-Ax}}$$

Al variare del parametro A la curva può diventare più o meno “ripida”. In particolare, la curva tende alla funzione a gradino $g(x)$ che abbiamo visto prima per $A \rightarrow -\infty$, mentre invece tende $g(-x)$ quando $A \rightarrow +\infty$.

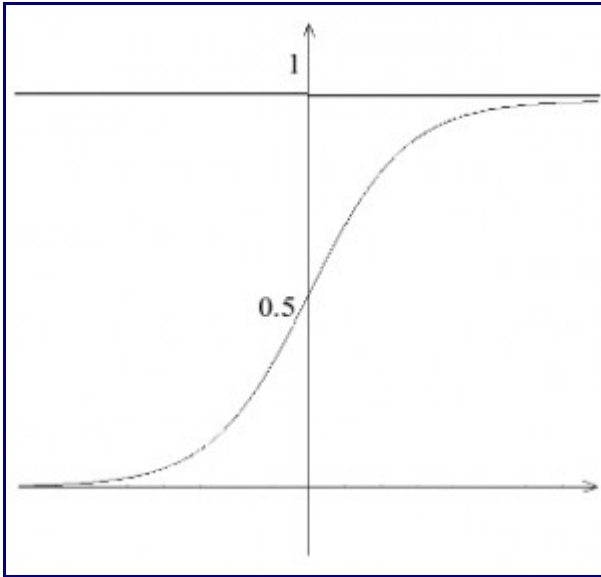


Grafico della curva sigmoideale

Se $x=0$, ovvero se $P=\theta$, allora il valore di uscita del neurone artificiale sarà 0.5, mentre invece sarà approssimativamente 0 (ovvero il neurone è “spento”) per $\theta \gg P$ e 1 per $\theta \ll P$. Una proprietà molto interessante di questa funzione, una proprietà molto utilizzata nella fase di apprendimento delle reti neurali, riguarda la sua derivata prima. In particolare

$$\frac{dy}{dx} = Ay(1 - y)$$

Questa proprietà implica che la derivata della funzione sigmoideale si può scrivere come un semplice prodotto, sorvolando le regole di derivazione, e questo è molto utile a fine computazionale (un calcolatore potrà trovare facilmente la derivata di una funzione così costruita).

Una funzione alternativa alla sigmoideale, relativamente meno usata nel campo delle reti neurali, è la tangente iperbolica, di equazione

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Una funzione definita $f : \mathbb{R} \rightarrow [-1, 1]$, a differenza delle due che abbiamo visto prima che sono a codominio in $[0,1]$.

Ogni neurone può dare in un dato momento, come abbiamo visto, un solo valore in output in funzione dei suoi input, mentre una rete neurale può complessivamente dare un numero variabile di valori in output. Se quindi una rete ha input n valori

x_1, x_2, \dots, x_n in un dato momento, la rete darà in output m valori y_1, y_2, \dots, y_m in quel momento, in funzione delle x_i . Ovvero

$$y_j = f_j(x_1, x_2, \dots, x_n) \text{ per } 1 \leq j \leq m$$

Quindi i valori in input in un certo momento sono un vettore X di n componenti, generalmente compresi tra 0 e 1. Anche gli m valori del vettore di output Y sono compresi tra 0 e 1, in quanto vengono confinati in questo intervallo dalla funzione di trasferimento usata (funzione a gradino o sigmoidale), quindi il vettore X va a identificare un punto A all'interno di un ipercubo booleano di n dimensioni, e Y un punto B all'interno di un'altro ipercubo booleano a m dimensioni. Nel caso di $n=m=2$ gli ipercubi degenerano in 2 quadrati di $2^n = 2^m = 4$ vertici, mentre invece nel caso $n=m=3$ gli ipercubi degenerano in 2 cubi di $2^n = 2^m = 8$ vertici. La rete neurale può quindi essere vista come un'applicazione binaria che associa a ogni punto A contenuto nel primo ipercubo un punto B contenuto nel secondo. La grande idea alla base delle reti neurali però non è solo l'applicazione binaria tra i punti di un insieme e i punti di un altro insieme. L'applicazione associa il punto A e anche un suo intorno ad un intorno del punto B del secondo insieme. Questo è molto utile nel caso in cui i segnali di input sono "sporcati", ad esempio nel caso di una rete neurale per il riconoscimento vocale, in grado di fare il suo dovere anche quando il suono è sporcato da rumori esterni, oppure una rete neurale per il riconoscimento calligrafico, in grado di fare il suo dovere anche quando il simbolo grafico non è perfettamente identico a quello appreso in fase di training. Questa proprietà deriva proprio dalle proprietà tipicamente fuzzy delle reti neurali.

Tecniche di apprendimento

Le reti neurali possono apprendere in due modi diversi: in modo supervisionato e in modo non supervisionato.

Nel primo caso, in ogni istante t vengono presentati alla rete dei campioni x_i in input, e i corrispondenti valori di output d_j desiderati. Le variazioni dei pesi sinattici

$$\Delta w = w(t + 1) - w(t)$$

sono una funzione dell'errore, e quindi dello scarto $y_j - d_j$, dove y_j è l'output ottenuto, e d_j l'output desiderato. Gli algoritmi di apprendimento in genere hanno come obiettivo quello di minimizzare l'errore quadratico medio. Quindi un'apprendimento supervisionato richiede la conoscenza sia dei valori di input x_i , sia dei valori desiderati d_j . Questi due tipi di dato forniscono quello che viene definito il training set della rete neurale.

Nel caso dell'apprendimento non supervisionato, vengono forniti alla rete molti campioni di input $X_i = (x_{i1}, \dots, x_{in})$, da associare a un numero m di classi

C_1, \dots, C_m . Il programmatore non fornisce alla rete la classe di appartenenza di ogni vettore di input; è la rete stessa ad auto-organizzarsi, modificando i suoi pesi sinattici in modo da poter eseguire classificazioni corrette. Gli algoritmi di apprendimento hebbiani sono classificabili all'interno di questa categoria. Questi algoritmi, basati sulla legge di Hebb, rafforzano il peso sinattico w_{ij} tra due generici neuroni i, j quando la loro attività è concorde (ovvero quando i risultati delle loro funzioni di trasferimento y_i, y_j sono di segno concorde), mentre lo indebolisce nel caso opposto, esattamente come accade tra i neuroni del sistema nervoso umano:

$$\Delta w_{ij} = \eta y_i y_j \text{ con } 0 \leq \eta \leq 1$$

Dove η è un coefficiente compreso tra 0 e 1 da cui dipende la variazione del peso sinattico.

Sviluppo di una rete neurale

Vediamo ora come implementare un'elementare rete neurale sfruttando algoritmi in C. La rete neurale che intendiamo sviluppare è relativamente semplice, ed è addestrata per svolgere la somma algebrica tra due numeri. Anche la sua funzione di attivazione è semplice (per questo esempio non useremo né la funzione a gradino né la sigmoide viste precedentemente, nonostante siano queste le funzioni più utilizzate), useremo la funzione identità $y=f(x)=x$ (ma potremmo usare una funzione qualsiasi, anzi ve lo lascio come esercizio).

Cominciamo a definire, magari in un file header, i tipi di dato di cui abbiamo bisogno:

```
typedef struct TypeNeuron neuron;
typedef struct TypeSynapsis sinapsi;
typedef struct TypeLayer layer;
typedef struct TypeNN neuralnet;
```

Ovvero i tipi di dati strutturati per i neurodi, le sinapsi, i layer e la rete neurale. Definiamole in questo modo:

```
#define _PRECISION float

struct TypeNeuron {
    _PRECISION trans_value;
    _PRECISION prop_value;
    sinapsi* in_links[16];
    int num_in_links;
    sinapsi* out_links[16];
    int num_out_links;
    _PRECISION (*trans_func)(_PRECISION prop_value);
};
```

Come valore di precisione della rete ho deciso di usare *float* (precisione semplice a

virgola mobile), ma nulla ci impedisce di usare int, long int o double. Le variabili *prop_value* e *trans_value* non identificano altro che, rispettivamente, il valore di propagazione x_i del neurone e il suo valore di trasferimento, ovvero il valore prodotto in output dalla funzione di trasferimento. Per il resto, dichiaro 16 collegamenti in entrata e 16 in uscita (ovvero 16 sinapsi che collegano il neurone ad altri 16 neuroni in ingresso e altre 16 che collegano il neurone a 16 neuroni in uscita), e tengo il conto delle sinapsi rispettivamente nelle variabili *num_in_links* e *num_out_links*. Infine dichiaro un puntatore a funzione (**trans_func*), in modo da poter scegliere successivamente che funzione di trasferimento usare per quello specifico neurone.

```
struct TypeSynapsis {
    _PRECISION delta;
    _PRECISION weight;
    neuron *in,*out;
};
```

Qui dichiaro il tipo sinapsi, caratterizzato da un puntatore al neurone di ingresso e uno a quello di uscita (in e out), un suo peso sinattico *weight* (corrispondente al w_i che abbiamo considerato finora nelle formule) e una sua delta, corrispondente alla variazione dei pesi sinattici in fase di apprendimento della legge di Hebb.

```
struct TypeLayer {
    neuron** elements;
    int num_elements;

    void (*update_weights)(layer* lPtr);
};
```

Un layer consiste in un insieme di neuroni, e conterrà quindi un puntatore alla lista di neuroni che ne fanno parte (*elements*), il loro numero (*num_elements*) e un puntatore a una funzione per aggiornare i pesi sinattici nella fase di apprendimento (*update_weights*). Una rete neurale semplice è generalmente composta di 3 layer:

- un layer di input che prende in ingresso i dati
- un layer di output che fornisce i dati elaborati
- uno o più layer nascosti (nel nostro caso ne basta uno) che connettono i due layer di input e output. Sono deputati alla fase di elaborazione dei dati

```
struct TypeNN {
    int max_epochs;
    _PRECISION l_rate;

    layer*    input_layer;
    layer*    hidden_layer;
    layer*    output_layer;
};
```

Questa struttura identifica la rete nel suo complesso, con i puntatori ai 3 layer che la compongono e due parametri che useremo in fase di apprendimento. In particolare,

max_epochs è il numero massimo di epoche, ovvero di cicli di aggiornamento dei pesi sinattici, che la rete può effettuare, mentre l_rate è il learning rate della rete (corrispondente all' η che abbiamo visto negli algoritmi di apprendimento).

Veniamo ora alle funzioni per inizializzare gli elementi della nostra rete:

```
void init_net(neuralnet *net) {
net = (neuralnet*) malloc(sizeof(neuralnet));
max_epochs=1024; // Valore arbitrario
l_rate=0.5; // Valore arbitrario
}
```

```
void new_layer(layer *l) {
l = (layer*) malloc(sizeof(layer));
num_elements=0;
}
```

```
void new_neuron(neuron *n) {
n = (neuron*) malloc(sizeof(neuron));
num_in_links=0;
num_out_links=0;
}
```

e ora una funzione per collegare tra di loro i layer:

```
void link_layers(layer* layer_in,layer* layer_out){
int i,j;
sinapsi* aux_syn;
neuron *curr_in,*curr_out;

for(i=0;i < layer_in->num_elements;i++) {
curr_in = layer_in->elements[i];

for(j=0;j < layer_out->num_elements; j++) {
curr_out = layer_out->elements[j];
aux_syn = (sinapsi*)malloc(sizeof(sinapsi));
aux_syn->in = curr_in;
aux_syn->out = curr_out;
aux_syn->weight = norm(get_rand());
curr_in->out_links[curr_in->num_out_links++] = aux_syn
curr_out->in_links[curr_out->num_in_links++] = aux_syn;
}
}
}
```

Questa funzione collega tra di loro due layer (layer_in e layer_out). Per fare ciò alloca memoria per ogni sinapsi tra ogni neurone di layer_in e ogni neurone di layer_out, attraverso due cicli for (il primo cicla su tutto il layer di input e il secondo su tutto il layer di output). Per ogni collegamento neurone-neurone viene creata una sinapsi, una sinapsi che, ovviamente, dovrà sapere che neuroni collegare, quindi:

```
aux_syn->in = curr_in;
```

```
aux_syn->out = curr_out;
```

e i neuroni stessi dovranno sapere che sinapsi utilizzare per collegarsi:

```
curr_in->out_links[curr_in->num_out_links++] = aux_syn  
curr_out->in_links[curr_out->num_in_links++] = aux_syn;
```

Per inizializzare il peso della sinapsi ho utilizzato un valore casuale, così calcolato dalla funzione `get_rand()`:

```
float get_rand() {  
    float x,y;  
  
    srand( (unsigned) time(NULL));  
    x = (float) rand();  
    y = (sin(x)*sin(x))-0.5;  
  
    return y;  
}
```

Quello che faccio è inizializzare il seme dei numeri casuali e assegnare alla variabile `x` un numero casuale. Per portare questo valore all'interno dell'intervallo $[-0.5, 0.5]$ sfrutto uno stratagemma matematico. Il codominio della funzione seno è in $[-1,1]$, quindi il codominio della funzione \sin^2x sarà ovviamente in $[0,1]$. Se sottraggo 0.5 al valore di questa funzione ottengo un valore compreso tra $[-0.5, 0.5]$.

Passiamo ora agli input da fornire alla rete. In questo esempio, forniremo alla rete degli input tramite un file in cui sono salvati dei numeri separati da ';'. La nostra rete dovrà imparare ad effettuare la somma algebrica, quindi nel file i primi due numeri rappresentano le quantità da sommare, e il terzo numero il risultato desiderato. Esempio:

```
1;2;3;5;7;12;3;5;8;.....
```

Dapprima dichiariamo una funzione che apra il file in questione in modalità lettura:

```
#define TRAINING_FILE "training.txt"  
  
int open_training_file() {  
    int fd;  
  
    if ((fd=open(TRAINING_FILE, O_RDONLY)) < 0)  
        return -1;  
    else  
        return fd;  
}
```

Vediamo ora la funzione `int get_data(float *data, int fd)`. Questa funzione prende come parametri un puntatore a float (in cui verrà salvato il numero letto) e un file descriptor (ottenuto dalla funzione `open_training_file()`), e ritorna -1 in caso di errore, altrimenti salva in `*data` il numero letto dal file fino al successivo ';'.

```

int get_data(float *data, int fd) {
int curr_char;
int status;
int is_dec;
char buf[1],ch;

// Attenzione: così come è dichiarata questa stringa può essere
// soggetta
// a buffer overflow. Imponete voi dei controlli ulteriori per
// evitarlo,
// controllando prima quanti caratteri ci sono nel file fino al
// prossimo
// ';' e dichiarando la stringa dinamicamente
char aux_str[256];

curr_char=0;

// Ciclo finché ci sono caratteri da leggere nel file
while( ( status = read(fd,buf,sizeof(buf)) ) != 0) {
    // Se status < 0, c'è qualche errore
    if(status<0)if(_DEBUG)perror(strerror(errno));
    ch=buf[0];

    // Se il carattere letto è proprio un ';', esco dal ciclo
    if(ch == ';') break;

    // Altrimenti continuo. Gli a capo sono ininfluenti
    if(ch == '\n')continue;

    // Gli unici caratteri validi al fine della lettura sono .
- e tutti // i valori numerici. Se il carattere letto non è uno di
quelli,
    // ritorno errore
    if(ch != '.' && ch != '-' && ( ch < 48 || ch > 57 )) {
        if(_DEBUG)fprintf(stderr,"invalid ch %d\n",ch);
        data = NULL;
        return -1;
    }

    // Controllo quanti punti ci sono nel numero
    if( ch == '.' ){
        // Se è già stato trovato un . allora c'è un errore
        if(is_dec){
            aux_str[curr_char++]=ch;
            aux_str[curr_char]='\0';
            fprintf(stderr,"invalid format: two '.' found in
%s\n",aux_str);
            return -1;
        }
    }

    // Altrimenti, il numero è decimale
    else

```

```

        is_dec=1;
    }

    // Salvo l'ulteriore carattere letto nella stringa aux_str
    aux_str[curr_char++] = ch;
}

// Termino la stringa
aux_str[curr_char]='\0';

// Converto la stringa in float e salvo il valore in *data
*data = atof(aux_str);
return 0;
}

```

Il codice è già abbastanza commentato, quindi non mi dilungherò ulteriormente. A questo punto studiamo il modo in cui la rete processa l'output. Quando la rete legge i valori di input, i neuroni del layer di input cambiano di valore, e i nuovi valori che assumono sono quelli della coppia di numeri. A questo punto, l'informazione passerà ai neuroni del layer nascosto, che calcoleranno prima il potenziale post-sinattico quindi il valore di uscita della funzione di trasferimento che, nel nostro caso, è una semplice funzione del potenziale post-sinattico. In particolare, avendo scelto come funzione di trasferimento la funzione identità, avremo

$$y = f(x) = x$$

Per cominciare, facciamo leggere gli input al layer di ingresso:

```

int fd;
int status;
float temp;

// Apro il file con gli input
fd=open_training_file();

// Ciclo su tutti gli elementi del layer di input
for(i=0; i < net->input_layer->num_elements; i++) {
    // Se la funzione get_data ritorna un valore negativo, allora
    c'è qualcosa
    // che non va negli input
    if((status = get_data(&temp,fd)) < 0){
        fprintf(stderr,"Invalid input data\n");
        free(net);
        return -1;
    }

    // Il valore del potenziale post-sinattico del neurone è quello
    appena
    // letto da input, e il valore di trasferimento sarà uguale in
    virtù della
    // scelta di funzione di trasferimento che abbiamo fatto
    net->input_layer->elements[i]->prop_value=(_PRECISION)temp;
}

```

```

    net->input_layer->elements[i]->trans_value=(_PRECISION)temp;
}

```

Per quanto riguarda invece il layer nascosto

```

void propagate_into_layer(layer* lPtr){
    int i;
    neuron* nPtr;

    // Ciclo for su tutti gli elementi del layer
    for(i=0;i < lPtr->num_elements;i++) {
        nPtr = lPtr->elements[i];
        // Per ogni neurone calcolo il potenziale post-sinattico...
        nPtr->prop_value = potential(nPtr);
        // ...e la funzione di trasferimento
        nPtr->trans_value = nPtr->trans_func(nPtr->prop_value);
    }
}

```

La funzione potential() ha questo codice:

```

_PPRECISION potential(neuron* nPtr){
    _PPRECISION aux_value=0;
    int i=0;

    // Per ogni sinapsi in ingresso al neurone...
    for(i=0; i<nPtr->num_in_links; i++) {
        // ...il valore del potenziale è la sommatoria del peso
        // sinattico della
        // sinapsi in questione moltiplicato per il suo valore di
        // trasferimento
        aux_value += (nPtr->in_links[i]->weight * nPtr->in_links[i]-
        >in->trans_value);
    }
    return aux_value;
}

```

Per i neuroni appartenenti al layer di output, il discorso è esattamente lo stesso fatto con il layer nascosto, e il codice rimarrà perfettamente identico.

A questo punto, abbiamo già visto che è possibile rendere più preciso il valore di uscita della rete neurale agendo sui singoli pesi sinattici. Per la struttura che abbiamo dato al file di input, la rete legge dal file sia i valori da sommare sia il risultato esatto, quindi provvederemo a far leggere il risultato giusto alla rete con la funzione get_data(). Una volta che abbiamo sia il risultato desiderato, sia il risultato effettivo della rete, opereremo sui pesi sinattici della rete in questo modo:

$$\Delta w_{ij} = -\eta D_j x_i$$

Dove η è una costante della rete compresa tra 0 e 1 chiamata learning rate, ed è definita a nostro piacimento (più il valore di η è alto, più la rete modificherà sensibilmente i suoi pesi sinattici in seguito a un errore). Un learning rate alto renderà

più veloce l'apprendimento della rete a discapito della precisione, mentre invece un learning rate basso renderà l'apprendimento più lento, ma la rete guadagnerà in fatto di precisione (diciamo pure che un valore intorno a 0.5 rappresenta un buon compromesso). x_i è il valore in input al neurone e D_j (delta di output) è così definita:

$$D_j = (y_j - d_j) \frac{d}{dx} f(P_j)$$

dove y_j e d_j sono rispettivamente il valore ottenuto in output e il valore desiderato, ed $f(P_j)$ è la funzione di trasferimento calcolata nel potenziale post-sinattico P_j . Questa è la base dell'algoritmo di apprendimento Widrow-Hoff, un algoritmo di apprendimento supervisionato che calcola i pesi necessari partendo da pesi casuali, e apportando a questi delle modifiche progressive in modo da convergere alla soluzione finale.

Nel nostro caso, poiché

$$y = f(P) = P \Rightarrow \frac{d}{dx} f(P) = 1$$

avremo semplicemente

$$\Delta w_i = y_i - d_i$$

La variazione verrà calcolata così:

```

PRECISION compute_output_delta(
PRECISION output_prop_value, PRECISION des_out) {
    PRECISION delta;

    delta =
(output_prop_value - des_out) * linear_derivate (output_prop_value);

    return delta;
}

```

dove `des_out` è il valore desiderato in output e `linear_derivate()` sarà, nel nostro caso, una funzione che ritornerà sempre 1 (ovviamente cambiando la funzione di trasferimento cambierà anche questa funzione).

Per aggiornare i pesi useremo l'equazione appena esaminata:

```

void update_output_weights(layer* lPtr, PRECISION delta, PRECISION
l_rate){
    int i,j;
    sinapsi* sPtr;
    neuron* nPtr;

```



```

for(i=0; i<lPtr->num_elements; i++) {
    nPtr = lPtr->elements[i];
    for(j=0; j < nPtr->num_in_links; j++){
        sPtr = nPtr->in_links[j];
        //  $\Delta w_{ij} = -\eta D_j x_i$ 
        sPtr->delta = -(sPtr->in->trans_value*delta*l_rate);
    }
}
}

```

Lo stesso algoritmo sarà valido anche per il layer nascosto. Ora, trovato l'incremento (o decremento) da applicare ai pesi sinattici, basterà ciclare su tutta la rete e applicare a tutte le sinapsi i nuovi pesi:

```

commit_weight_changes(net->output_layer);
commit_weight_changes(net->hidden_layer);

```

con

```

void commit_weight_changes(layer* lPtr){
    int i,j;
    neuron* nPtr;
    sinapsi* sPtr;

    // Ciclo su tutti gli elementi del layer
    for(i=0; i < lPtr->num_elements; i++) {
        nPtr = lPtr->elements[i];

        // Ciclo su tutte le sinapsi collegate ad un certo neurone
        for(j=0; j < nPtr->num_in_links; j++) {
            // La sinapsi sarà associata al j-esimo collegamento del
            neurone
            sPtr = nPtr->in_links[j];

            // Il peso della sinapsi viene aggiornato con il delta
            // appena calcolato
            sPtr->weight += sPtr->delta;

            // Resetto il valore di delta, in modo da potergli
            applicare
            // nuove modifiche
            sPtr->delta = 0;
        }
    }
}

```

Come intuibile, maggiore sarà il numero di epoche (ovvero di iterazioni di questo tipo, in cui modifico il valore dei pesi per convergere sempre più al valore desiderato), maggiore sarà la precisione dei valori di output della rete. Il valore massimo di iterazioni ammissibile l'avevamo precedentemente stabilito all'interno della variabile `max_epochs`.

A questo punto, nel nostro main() inseriamo un ciclo che effettua automaticamente questo procedimento per max_epochs volte:

```
// Ciclo per max_epochs volte
for(j=0; j<net->max_epochs; j++) {
    // Leggo i valori in input dal file, con il procedimento già
    visto
    // in precedenza
    for(i=0; i<net->input_layer->num_elements; i++) {
        if((status = get_data(&temp,fd)) < 0){
            fprintf(stderr,"errore irreversibile, closing...\n");
            free(net);
            return -1;
        }
        net->input_layer->elements[i]->prop_value=(_PRECISION)temp;
        net->input_layer->elements[i]->trans_value=(_PRECISION)temp;
    }

    // Passo i valori prima al layer nascosto, quindi al layer di
    output
    propagate_into_layer(net->hidden_layer);
    propagate_into_layer(net->output_layer);

    // Calcolo la delta di output
    if((status = get_data(&des_out,fd)) < 0){
        fprintf(stderr,"errore irreversibile, closing...\n");
        return -1;
    } else {
        out_delta = compute_output_delta(net->output_layer-
        >elements[0]->prop_value,des_out);
        update_output_weights(net->output_layer,out_delta,net-
        >l_rate);
    }

    // Calcolo la variazione dei pesi sinattici per il layer
    nascosto
    // e aggiorno tutti i pesi sinattici
    update_hidden_weights(net->hidden_layer,out_delta,net->l_rate);
    commit_weight_changes(net->output_layer);
    commit_weight_changes(net->hidden_layer);
    net_output = net->output_layer->elements[0]->prop_value;
    printf("DES=%f\tERROR=%f\tOUT=%f\tDELTA=%f\n",des_out,
        (des_out-net_output),net_output,out_delta);
}
```

Ed ecco che la nostra rete neurale è pronta per l'uso.

Riferimenti bibliografici

- Silvio Cammarata - Sistemi fuzzy
- Silvio Cammarata - Reti neurali

- M.Latini, P.Lulli - Imparare il C: una guida per Linux (che ringrazio per gli ottimi riferimenti di codice in C, una guida alla programmazione semplicemente eccellente)

Raw socket

.

I socket standard usati in C sono relativamente comodi da usare in quanto automatizzano tutti i meccanismi implementati dal protocollo TCP/IP, lasciando allo sviluppatore solo la responsabilità del livello applicativo. Tuttavia in alcuni contesti si vuole avere il controllo completo di ciò che viene inviato sulla rete. Applicazioni tipiche sono l'[IP spoofing](#) (invio di un pacchetto ad un host con un altro IP) e i conseguenti attacchi [Smurf](#). In questi casi può risultare comodo costruirsi il pacchetto inviato sull'interfaccia di rete pezzo per pezzo. Per fare ciò il C mette a disposizione i *raw socket*, dei socket su cui è possibile inviare pacchetti *grezzi* creati dallo sviluppatore (ovviamente delle profonde conoscenze dei protocolli di rete e di trasporto sono richieste). Vediamo subito un esempio pratico con un'applicazione che crea un pacchetto da zero che pinga localhost e lo invia su raw socket:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <linux/ip.h>

#define ICMP_ECHO          8
#define IPLEN              sizeof(struct iphdr)
#define ICMPLEN            sizeof(struct icmphdr)

typedef unsigned char  u8;
typedef unsigned short u16;
typedef unsigned long  u32;

struct icmphdr {
    u8          type;
    u8          code;
    u16         checksum;
    u16         id;
    u16         sequence;
};

unsigned short csum (u16 *buf, int nwords) {
    unsigned long sum;

    for (sum = 0; nwords > 0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return ~sum;
}
```

```

}

main() {
    int i,sd,one,len;
    unsigned char buff[BUFSIZ],in[BUFSIZ];
    char data[56];
    char *tmp;

    struct sockaddr_in sin;
    struct iphdr *ip = (struct iphdr*) malloc(IPLLEN);
    struct icmp_hdr *icmp = (struct icmp_hdr*) malloc(ICMPLEN);

    srand ((unsigned) time(NULL));
    sd=socket (PF_INET, SOCK_RAW, IPPROTO_ICMP);

    sin.sin_family=AF_INET;
    sin.sin_port=0;
    sin.sin_addr.s_addr=inet_addr("127.0.0.1");

    memset (buff,0,sizeof(buff));

    for (i=0; i<56; i++)
        data[i]=i;

    ip->version=4;
    ip->ihl=5;
    ip->tos=0;
    ip->tot_len=IPLLEN+ICMPLEN+sizeof(data);
    ip->id=0;
    ip->frag_off=0;
    ip->ttl=64;
    ip->protocol=IPPROTO_ICMP;
    ip->check=0;
    ip->saddr=inet_addr("127.0.0.1");
    ip->daddr=inet_addr("127.0.0.1");
    ip->check = csum ((u16*) buff, ip->tot_len >> 1);

    icmp->type=ICMP_ECHO;
    icmp->code=0;
    icmp->checksum=0;
    icmp->id=1;
    icmp->sequence=1;

    tmp = (char*) malloc(ICMPLEN+sizeof(data));
    memcpy (tmp, icmp, ICMPLEN);
    memcpy (tmp+ICMPLEN, data, sizeof(data));
    icmp->checksum=csum((u16*) tmp, ICMPLEN+sizeof(data) >> 1);

    memcpy (buff, ip, IPLLEN);
    memcpy (buff+IPLLEN, icmp, ICMPLEN);
    memcpy (buff+IPLLEN+ICMPLEN, data, sizeof(data));

    one=1;

```

```

        if (setsockopt (sd, IPPROTO_IP, IP_HDRINCL, &one, sizeof
(one)) < 0)
            printf ("Warning: Cannot set HDRINCL!\n");

        if (sendto (sd,
                    buff,
                    ip->tot_len,
                    0,
                    (struct sockaddr *) &sin,
                    sizeof (sin)) < 0) {
            printf ("Error in send\n");
            exit(1);
        } else
            printf ("Send OK\n");
    }
}

```

Vediamo i componenti notevoli:

```
#include <linux/ip.h>
```

Questa dichiarazione è necessaria per poter usare la struttura *iphdr*, contenente tutti i campi di un header IP, che semplifica notevolmente il lavoro. Successivamente dichiaro la struttura di un header ICMP (*icmphdr*).

```

unsigned short csum (u16 *buf, int nwords) {
    unsigned long sum;

    for (sum = 0; nwords > 0; nwords--)
        sum += *buf++;
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    return ~sum;
}

```

Questa è la funzione per il calcolo del checksum di un header (complemento a 1 della somma dei complementi a 1 dell'header diviso in word da 16 bit). In seguito inizializzo il socket come socket raw

```
sd=socket (PF_INET, SOCK_RAW, IPPROTO_ICMP);
```

riempio gli ultimi 56 byte del pacchetto con byte casuali (struttura classica di un pacchetto ping)

```

for (i=0; i<56; i++)
    data[i]=i;

```

quindi riempio gli header IP e ICMP:

```

ip->version=4;
ip->ihl=5;
ip->tos=0;

```

```

ip->tot_len=IPLen+ICMPLEN+sizeof(data);
ip->id=0;
ip->frag_off=0;
ip->ttl=64;
ip->protocol=IPPROTO_ICMP;
ip->check=0;
ip->saddr=inet_addr("127.0.0.1");
ip->daddr=inet_addr("127.0.0.1");
ip->check = csum ((u16*) buff, ip->tot_len >> 1);

icmp->type=ICMP_ECHO;
icmp->code=0;
icmp->checksum=0;
icmp->id=1;
icmp->sequence=1;

```

A questo punto effettuo il calcolo del checksum ICMP

```

tmp = (char*) malloc(ICMPLEN+sizeof(data));
memcpy (tmp, icmp, ICMPLEN);
memcpy (tmp+ICMPLEN, data, sizeof(data));
icmp->checksum=csum((u16*) tmp, ICMPLEN+sizeof(data) >> 1);

```

in quanto dovrò calcolare il checksum sull'header ICMP e sulla parte di dati. Ora copio le strutture così riempite in un buffer

```

memcpy (buff, ip, IPLen);
memcpy (buff+IPLen, icmp, ICMPLEN);
memcpy (buff+IPLen+ICMPLEN, data, sizeof(data));

```

setto l'opzione IP_HDRINCL sul socket (necessaria per iniettare pacchetti raw, richiede i privilegi di root)

```
one=1;
```

```

if (setsockopt (sd, IPPROTO_IP, IP_HDRINCL, &one, sizeof
(one)) < 0)
    printf ("Warning: Cannot set HDRINCL!\n");

```

quindi effettuo l'invio tramite sendto:

```

if (sendto (sd,
            buff,
            ip->tot_len,
            0,
            (struct sockaddr *) &sin,
            sizeof (sin)) < 0) {
    printf ("Error in send\n");
    exit(1);
} else
    printf ("Send OK\n");

```

Monitorare modifiche ai file tramite inotify

Il kernel Linux mette a disposizione un mezzo estremamente potente per monitorare le modifiche di qualsiasi tipo a file e directory: l'oggetto *inotify*. Le potenzialità e la comodità sono non indifferenti: la possibilità è quella, ad esempio, di controllare se un log di sistema viene aggiornato con dei nuovi eventi, e inviare ad esempio questi messaggi di log direttamente all'email dell'amministratore. Oppure controllare se i file in una directory riservata vengono modificati, e gestire l'evento come si vuole. L'header da includere è

```
#include <linux/inotify.h>
```

quindi la procedura è

- Inizializzare *inotify* (*inotify_init()*)
- Aggiungere un file o una directory su cui effettuare il *watch* (*inotify_add_watch()*)
- Controllare in un ciclo se vengono effettuati cambiamenti sul file o la directory in questione
- Gestire i cambiamenti come si vuole
- Rimuovere il *watch_point* (*inotify_rm_watch()*)

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <linux/inotify.h>
```

```
#ifndef BUFSIZ
#define BUFSIZ 8139
#endif
```

```
int main (int argc, char **argv) {
    char buff[BUFSIZ];
    int fd, // Descrittore del file
        ifd, // Descrittore dell'istanza di inotify
        wd, // Descrittore del watch
        n; // Numero di byte letti

    time_t ltime;
    char *strtime;
```



```

// Se non c'è nessun argomento passato, esco
if (!argv[1])
    return 1;

// Apro il file passato come primo argomento
if ((fd=open(argv[1], O_RDONLY))<0)
    return 2;

// Inizializzo inotify. inotify_init() ritornerà un nuovo
descrittore, che userò per gestire
// i watch point
if ((ifd=inotify_init())<0)
    return 3;

// Aggiungo un watch point che monitora le modifiche al file
specificato via argv[1]
wd = inotify_add_watch (ifd,argv[1],IN_MODIFY);

// Mi posiziono alla fine di argv[1]
lseek (fd,0,SEEK_END);

// Ciclo infinito
while(1) {
    // Leggo dal descrittore di inotify. La read è
    bloccante, quindi procede solo quando ci sono
    // byte da ricevere da ifd, ovvero solo quando viene
    effettuata una modifica su argv[1]
    read (ifd,buff,BUFSIZ);

    // Leggo l'ora e la data attuale
    ltime = time((unsigned) NULL);
    strtime = strdup(ctime(&ltime));
    strtime[strlen(strtime)-1]=0;

    // Contollo quanti byte sono stati aggiunti al file
    e
    while ((n=read(fd,buff,BUFSIZ))>0)
        printf ("[%s] %s modified: %s\n",strtime,
argv[1], buff);
    }

    // Rimuovo il watch point (non arriverà mai qui)
    inotify_rm_watch (ifd,wd);
}

```

Quello che faccio è monitorare le modifiche effettuate al file passato via argv. Quindi apro il file in questione in modalità lettura, posizionandomi alla sua fine, e dico al mio oggetto inotify di monitorare le modifiche che vengono effettuate a quel file. *IN_MODIFY* è solo uno dei possibili eventi che posso monitorare usando inotify. L'elenco completo è il seguente:

```

IN_ACCESS      File was accessed (read) (*).

```

extended	IN_ATTRIB	Metadata changed (permissions, timestamps, attributes, etc.)
		(*).
	IN_CLOSE_WRITE	File opened for writing was closed (*).
	IN_CLOSE_NOWRITE	File not opened for writing was closed (*).
	IN_CREATE	File/directory created in watched directory (*).
	IN_DELETE	File/directory deleted from watched directory
(*)		
	IN_DELETE_SELF	Watched file/directory was itself deleted.
	IN_MODIFY	File was modified (*).
	IN_MOVE_SELF	Watched file/directory was itself moved.
	IN_MOVED_FROM	File moved out of watched directory (*).
	IN_MOVED_TO	File moved into watched directory (*).
	IN_OPEN	File was opened (*).

Ora mi ritrovo in mano con un nuovo descrittore, associato all'istanza di inotify, e in un ciclo infinito leggo da quest'ultimo. Poiché la read ha effetto bloccante, potrò andare avanti solo quando ci sono byte da leggere da quel descrittore, ovvero quando è stata materialmente compiuta qualche modifica sul file specificato. In questo caso, leggo la data e l'ora della modifica, leggo i byte aggiunti e stampo le modifiche su stdout, o su un file di log, o dove voglio.

Questa riga

```
read (ifd, buff, BUFSIZ);
```

salva il risultato della read sul descrittore di inotify in un buffer in memoria. La read su un descrittore di inotify produce in realtà un'istanza della struttura *inotify_event*, definita in questo modo:

```
struct inotify_event {
    int      wd;          /* Watch descriptor */
    uint32_t mask;       /* Mask of events */
    uint32_t cookie;     /* Unique cookie associating related
                          events (for rename(2)) */
    uint32_t len;       /* Size of 'name' field */
    char     name[];    /* Optional null-terminated name */
};
```

dalla quale posso ad esempio vedere il descrittore del mio watch point che è stato invocato, la maschera che ho usato per filtrare gli eventi, il cookie associato all'evento e, se esiste, la stringa *name*, che nel caso di una directory *watched*, contiene il nome del file che è stato modificato, con la relativa lunghezza della stringa salvata in *len*.

*a rgod, per aver dimostrato
al mondo intero che in
Italia
è ancora possibile fare
hacking senza cadere nel
banale*

*a tutti coloro che hanno
creduto nella possibilità di
realizzare in Italia un
portale come BlackLight.es
e portare avanti progetti
ambiziosi come
HacKnowledge*