yaC-Primer

Yet Another C-Primer

A. Crisanti

Versione 2, 2003
Revisione 1 (in corso), 2005

Questo Primer è basato sulle lezioni tenute presso il Dipartimento di Fisica dell'Università di Roma "La Sapienza" nei corsi di *Laboratorio di Calcolo* e *Laboratorio di Fisica Computazionale I* per gli studenti del primo e secondo anno della Laurea triennale in Fisica.

Lo scopo principale di questo Primer è quello di insegnare non solo il linguaggio C ma anche la programmazione in linguaggio C. Non è richiesta tuttavia nessuna esperienza o conoscenza di programmazione precedente. Questo spiega la dovizia di particolari con cui sono discussi i programmi negli esempi svolti presentando a volte il programma completo nelle sue fasi evolutive di sviluppo oppure programmi completi che risolvono lo stesso problema con soluzioni diverse. Scopo di questo Primer è infatti anche quello di cercare di insegnare un metodo di programmazione e di risoluzione di problemi mediante l'uso di un computer. A scopo puramente didattico si suggerisce di provare a scrivere autonomamente il programma relativo ad un dato esempio prima di guardare il programma proposto.

Gli esempi discussi sono le prove di laboratorio proposte agli studenti nella parte pratica dei corsi e seguono lo sviluppo del Primer proponendo soluzioni adeguate alle conoscenze di Linguaggio C acquisite fino a quel momento, è quindi utile riconsiderare gli esercizi man mano che le conoscenze aumentano per confrontare le diverse possibilità offerte dal linguaggio C.

Il linguaggio C è un linguaggio *general purpose*, per cui può essere utilizzato per molti scopi differenti. In questo Primer l'enfasi è messa verso problemi e metodi utilizzati in fisica fornendo un primo approccio con la "fisica computazionale".

Indice

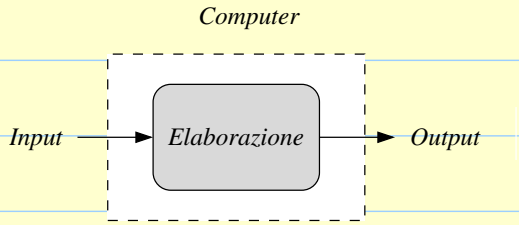
Part I.

Primer C

1. Computers

1.1. Computers Analogici e Digitali (Rev. 2.1)

La funzione principale di un computer è quella di *elaborare* i dati forniti in ingresso ("*input*") e produrre un risultato in uscita ("*output*"). Possiamo quindi definire genericamente un computer come un qualsiasi sistema che prende dei dati in input, li elabora e produce un risultato in output.



I computer si dividono in due grandi classi: computers *analogici* e computers *digitali*.

Computer Analogici

Nei computers analogici i dati sono rappresentati da grandezze fisiche che possono variare in maniera *continua*, come ad esempio la pressione, la posizione o il voltaggio.

I computers analogici eseguono le operazioni mediante circuiti elettronici il cui funzionamento riproduce l'operazione voluta. Ad esempio la somma di due numeri può essere ottenuta mediante un circuito che ha come output la somma delle differenze di potenziale applicate in input.

I computer analogici sono spesso usati per simulare sistemi fisici descritti da equazioni differenziali anche complesse. Il principale svantaggio dei computer analogici è la loro scarsa versatilità. Un computer analogico viene infatti costruito per risolvere una ben determinata classe di problemi e difficilmente può essere usato per risolverne altri.

Computer Digitali

I computer digitali elaborano dati rappresentati da *digits* i cui valori dipendono dal tipo di digit usato. Ad esempio un digit decimale assume 10 differenti valori indicati dai simboli $0, 1, ..., 9$. Di conseguenza nei computer digitali i numeri non sono rappresentati mediante

grandezze che possono variare in maniera continua ma da sequenze di digits in una forma molto simile a quella scritta.

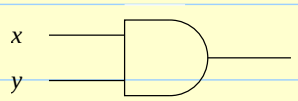
La stragrande maggioranza di computers digitali utilizza per la rappresentazione di un digit fenomeni fisici che hanno solo due stati stabili, di solito indicati come stato 0 e stato 1. Nella tavola seguente sono riportati alcuni esempi

Sistema	**Stato** 0	**Stato** 1
Tensione	$0 - 0.8$ V	$2.0 - 5.0$ V
Condensatore	Scarico	Carico
Transistor	Interdizione	Saturazione
Fibra ottica	Luce spenta	Luce accesa
Disco	Flusso magnetico non invertito	Flusso magnetico invertito

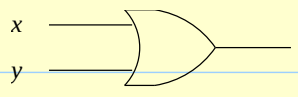
Un digit che può assumere solo due valori si chiama *digit binario* o *bit*. In generale, a meno che non sia detto esplicitamente, per computer digitale si intende un computer che utilizza digits binari.

Logica Binaria o Booleana

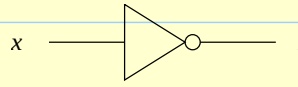
I computers digitali manipolano i bits utilizzando porte logiche (*gates*), realizzate mediante circuiti elettronici, che prendono come input uno o più bits e producono un bit di output. L'elaborazione dei bits avviene utilizzando la logica binaria o booleana, da George Boole, i cui gates principali sono i gates logici `AND`, `OR` e `NOT`:



x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1



x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1



x	NOT x
0	1
1	0

Ogni altro gate logico può essere costruito a partire da questi gates principali.

Memoria

I gates logici possono essere usati, oltre che per manipolare bits, anche per memorizzare il valore di un bit. Questi elementi di memoria sono ottenuti mediante gates logici chiamati *flip-flop* il cui output si può trovare sia nello stato stabile 0 che nello stato stabile 1 a seconda del valore del bit di input. Siccome lo stato è stabile il valore dell'output rimarrà inalterato fino a che un nuovo bit di input (o una mancanza di corrente) non intervenga a modificarne il valore, memorizzando così il valore dell'ultimo bit di input.

Un flip-flop può memorizzare il valore di un solo bit, di conseguenza per memorizzare lo stato di più bits si devono usare più flip-flops. Ad esempio 16 flip-flops formano un elemento di memoria in cui può essere memorizzato il valore di al massimo 16 bits. Il numero di bits che possono essere memorizzati definisce la dimensione della memoria, quindi nel nostro esempio precedente la dimensione è di 16 bits. Alcuni raggruppamenti di bits hanno nomi speciali:

$$\begin{array}{lcl} 4 \text{ bits} & \Rightarrow & \textit{nibble} \\ 8 \text{ bits} & \Rightarrow & \textit{byte} \end{array}$$

per cui avremmo potuto anche dire che la dimensione è di 2 bytes.

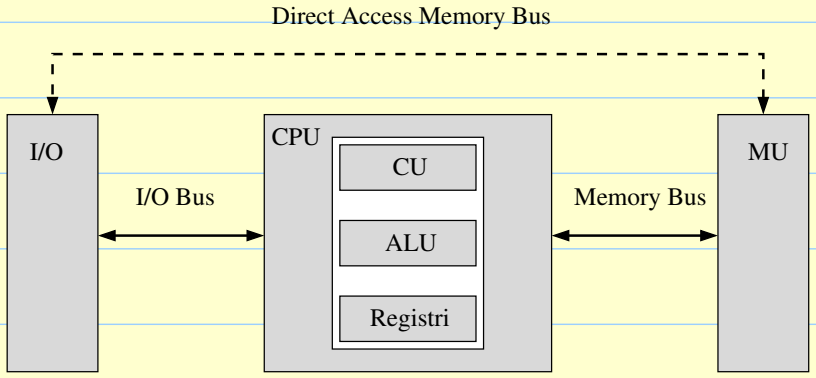
I bits della memoria sono suddivisi in piccoli gruppi, chiamati *registri* di memoria, usualmente formati da 8 bits ovvero 1 byte. Una unità di memoria può contenere molti registri, di conseguenza come per le usuali unità di misura per specificarne la dimensione si usano suffissi come "*kilo*", "*mega*" e così via. Tuttavia nel caso della memoria questi hanno un valore diverso da quello usuale:

Suffisso	Simbolo	Valore Usuale	Valore Memoria
kilo	K	10^3	$2^{10} = 1024$
mega	M	10^6	$2^{20} = 1048576$
giga	G	10^9	$2^{30} = 1073741824$

Per cui ad esempio 1 megabyte di memoria corrisponde a 1048576 bytes ovvero 8388608 bits.

1.2. Architettura di base di un computer digitale (Rev. 2.1)

Preso nel suo insieme un computer digitale è un insieme componenti elettronici, che vanno sotto il nome generico di *hardware*, organizzati in un'architettura su diversi livelli estremamente complessa. In generale i dettagli dell'architettura possono variare da un computer all'altro tuttavia, senza entrare in troppi dettagli, questa può essere divisa in tre sottosistemi principali connessi fra di loro:



CPU

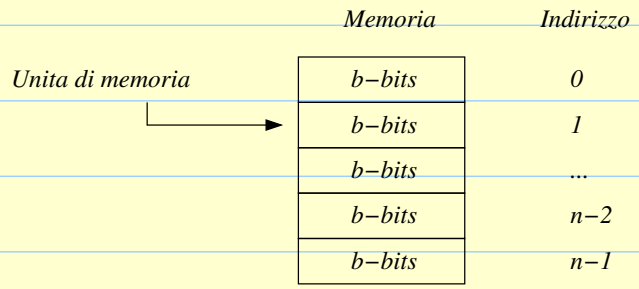
Il processore o CPU (*Central Process Unit*) è la parte del computer che si occupa dell'elaborazione ed é composto in genere da:

CU	:	Unità di controllo (*Control Unit*) per leggere ed eseguire le istruzioni e per trasferire i dati da e verso la memoria
ALU	:	Unità loigca/matematica (*Arithmetic/Logic Unit*) per manipolare i dati
Registri	:	Registri di memoria per memorizzare lo stato del processore e piccole quantità di dati

Il processore contiene anche dei circuiti per controllare e comunicare con la memoria (MU) e l'Input/Output (I/O).

MU

Il sottosistema di memoria MU (*Memory Unit*) contiene sia i dati che le istruzioni (*programmi*) che devono essere elaborati dalla CPU. La MU è formata da un insieme di *unità di memoria* ciascuna delle quali può contenere il valore di b-bits, è quindi come un insieme di cassetti in cui possono essere memorizzati e letti i valori dei bits. Le unità di memoria sono disposte sequenzialmente ed individuate da un indirizzo di memoria (*memory address*) che rappresenta la loro posizione nella sequenza, di conseguenza se la memoria contiene n unità di memoria l'indirizzo va da 0 per la prima unità di memoria fino a $n-1$ per l'ultima:



Siccome l'indirizzo individua univocamente una unità di memoria a volte ci si riferisce a quest'ultime semplicemente come indirizzi di memoria.

La CPU può accedere con la stessa velocità ad una qualsiasi unità di memoria semplicemente specificandone l'indirizzo, in altre parole la velocità di accesso non dipende dal valore dell'indirizzo. Questo tipo di accesso viene chiamato *Random Access* e per questo la memoria viene spesso chiamata *Random Access Memory* (RAM).

Questa è la caratteristica principale della MU che la distingue dagli altri tipi di memoria come ad esempio su disco, nastro magnetico o CD, in cui l'accesso è invece sequenziale per cui per accedere all'unità di memoria di indirizzo m bisogna partire dall'unità di memoria di indirizzo 0 e scorrerle tutte fino all'indirizzo voluto. L'accesso è quindi tanto più lento quanto più è grande l'indirizzo richiesto. La memoria sequenziale è come un quaderno: per accedere ad una pagina si devono sfogliare tutte le pagine del quaderno dalla prima pagina fino alla pagina desiderata.

I/O

Il sottosistema di Input/Output I/O è la parte del computer che si occupa delle "interazioni" del computer con "il mondo esterno" e comprende:

Periferiche	:	terminali, tastiere, comunicazioni (modem, LAN,...), stampanti, etc.
Memorie di Massa	:	dischi, unità nastri, CD/DVD, etc.

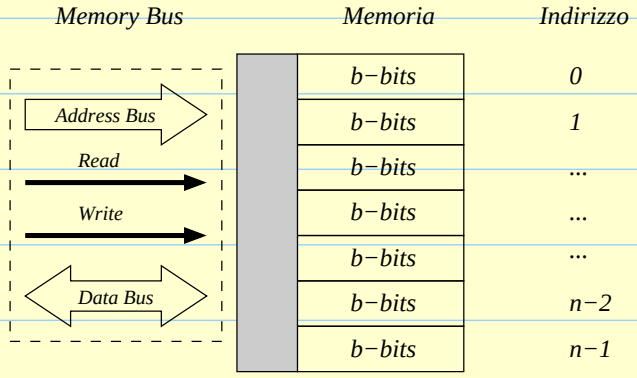
Normalmente le memoria di massa sono ad accesso sequenziale e quindi più lente della RAM.

Bus

I tre sottosistemi comunicano tra loro per lo scambio di istruzioni e dati mediante canali di comunicazione chiamati Bus:

Memory Bus	:	canale di comunicazione tra CPU e MU
I/O Bus	:	canale di comunicazione tra CPU e I/O
Direct Access Memory Bus	:	canale di comunicazione tra I/O e MU

A loro volta ciascun Bus può essere composto da più Busses. Ad esempio il Memory Bus comprende un Address Bus che contiene l'indirizzo di memoria RAM richiesto, un Read/Write Bus per specificare il tipo di operazione richiesta ed un Data Bus per lo scambio dei dati.



I primi due Busses sono unidirezionali poichè contengono informazioni dalla CPU alla MU, il Data Bus è invece bidirezionale in quanto se è attivo il Write Bus i dati vanno dalla CPU alla MU, mentre se è attivo il Read Bus i dati devono andare dalla MU alla CPU.

I diversi Busses non sono necessariamente fisicamente distinti, ed infatti in molti computer l'I/O Bus ed il Memory Bus sono lo stesso e la CPU non fa distinzione tra i due sottosistemi. Infine in alcuni computers, specialmente i più semplice, il Direct Access Memory Bus manca ed ogni comunicazione tra I/O e MU avviene attraverso la CPU.

1.3. Software di base di un computer digitale (Rev. 2.1)

Per *software* di un computer si intende l'insieme delle istruzioni e dati che il computer manipola per produrre qualche cosa di utile. Un programma è una sequenza di istruzioni per la CPU. Sia i programmi che i dati sono di solito memorizzati in oggetti chiamati *files* che risiedono nelle memorie di massa.

Come l'architettura di un computer anche il software è diviso in vari livelli. Partendo da livello più esterno l'organizzazione più semplice è:

Sistema Operativo
Linguaggi di Alto Livello
Assembler
Linguaggio Macchina

Linguaggio Macchina

Il Linguaggio Macchina consiste in istruzioni *primitive* che possono essere lette direttamente dal processore. Queste sono le uniche istruzioni che il processore capisce e chiaramente dipendono fortemente da quest'ultimo. Le istruzioni sono codificate come stringhe di bits chiamate parole (*words*). Come le istruzioni anche la lunghezza della parola, ossia il numero di bits che la compongono, dipende dal processore. I processori più diffusi usano

word di 16 bits ⇒ CPU a 16-bits
word di 32 bits ⇒ CPU a 32-bits
word di 64 bits ⇒ CPU a 64-bits

Per eseguire le istruzioni queste vengono prima caricate nella RAM, insieme agli eventuali dati necessari, da dove la CPU le legge una alla volta sequenzialmente partendo dall'indirizzo di memoria 0 il cui contenuto viene sempre interpretato come un'istruzione.

Il principale difetto del linguaggio macchina, oltre ad essere dipendente dalla CPU, è la sua difficile comprensione da parte degli utenti "umani". Inoltre la scrittura di programmi in linguaggio macchina richiede una conoscenza piuttosto dettagliata della struttura e dei registri della CPU.

Assembler

Il linguaggio Assembler risolve il problema della comprensione "umana" associando alle istruzioni del linguaggio macchina dei codici mnemonici più facilmente leggibili delle sequenze di bits, come mostrato nel seguente esempio. Per il resto l'Assembler è a tutti gli effetti equivalente al linguaggio macchina per cui tutti i programmi in linguaggio macchina sono di fatto scritti in Assembler.

Esempio: CPU 8-bits

Istruzione	**Linguaggio Macchina**	**Assembler**
Aggiungi il valore contenuto all'indirizzo di memoria 35 al registro B della CPU	`11011011 00100011`	`ADDB VAR`

dove `00100011` = 35 in rappresentazione 8-bits e `VAR` si assume contenga il valore 35.

La CPU non comprende le istruzioni del linguaggio Assembler, di conseguenza affinchè un programma scritto in Assembler possa essere eseguito dalla CPU questo deve essere prima trasformato in un programma con istruzioni in linguaggio macchina. Questo è ottenuto mediante un programma chiamato Assembler che traduce (*compila*) il programma dal linguaggio Assembler al linguaggio macchina.

I programmi scritti in Assembler sono molto veloci per questo l'Assembler è molto usato per scrivere programmi o parte di programmi che sono utilizzati molto frequentemente. Tuttavia

il linguaggio Assembler, pur essendo più comprensibile del linguaggio macchina rimane un linguaggio non molto trasparente ed inoltre, come il linguaggio macchina, fortemente dipendente dal processore.

Altro svantaggio dell'Assembler è la lunghezza dei programmi. L'Assembler usa istruzioni primitive per istruire il processore a compiere certe operazioni, di conseguenza operazioni semplici come ad esempio "*ripeti la seguente operazione fino a che X è minore di* 0" richiede molte istruzioni in Assembler. Chiaramente siccome il tempo per scrivere un programma la probabilità di fare errori aumenta con la lunghezza del programma è chiaro che l'Assembler non è la scelta migliore per la programmazione.

Linguaggi di Alto Livello

Per ovviare alle difficoltà dell'Assembler sono stati sviluppati linguaggi di programmazione più orientati verso il programmatore "umano" che verso il computer. Questi linguaggi di programmazione di alto livello, contrariamente all'Assembler, non dipendono dal tipo di processore ed inoltre permettono di eseguire operazioni anche complesse con poche istruzioni. Questo rende i programmi scritti in questi linguaggi non solo più facilmente comprensibili ma anche più portabili tra computers diversi.

Come avviene per i programmi scritti in Assembler anche i programmi scritti in un linguaggio di alto livello devono essere preventivamente compilati mediante un opportuno compilatore che traduca le istruzioni di alto livello nelle corrispondenti istruzioni in linguaggio macchina affinchè la CPU possa eseguirli. Siccome è il compilatore che si prende cura di trasformare le istruzioni di alto livello indipendenti dalla CPU nelle corrispondenti istruzioni primitive, differentemente dalla programmazione in Assembler, la programmazione in un linguaggio di altro livello non richiede la conoscenza dettagliata dell'architettura della CPU. Per contro un programma scritto in un linguaggio di alto livello è di solito più lento dello stesso programma scritto in Assembler. Inoltre siccome è il compilatore che fa la traduzione, la velocità del programma può dipendere sia dal compilatore usato che dallo stile di programmazione. Infatti a volte basta riscrivere parti del programma in una forma diversa, più (o meno) consona ai "desiderata" del compilatore, per diminuirne (o aumentarne!) la velocità di esecuzione.

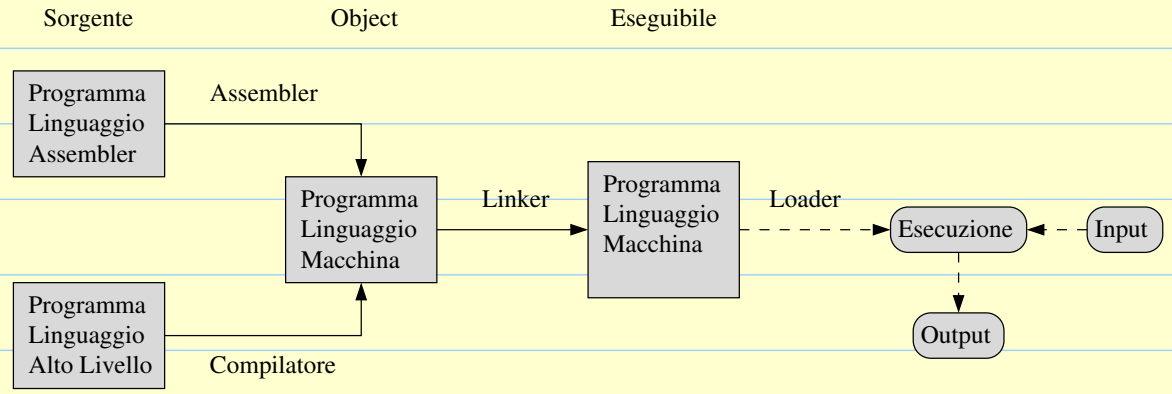
Compilatore e Linker

I compilatori sono essi stessi dei *programmi* che traducono un programma scritto in un linguaggio diverso dal linguaggio macchina in un programma equivalente in linguaggio macchina. Il programma prodotto dal compilatore si chiama *object*. Per la loro natura chiaramente i compilatori dipendono dalla CPU.

Sebbene l'object sia un programma scritto in linguaggio macchina, e quindi interpretabile dalla CPU, questo non può essere ancora eseguito dal computer. L'object è infatti la semplice traduzione in linguaggio macchina del programma scritto in un linguaggio in alto livello o in Assembler, ma affinchè il programma possa essere effettivamente eseguito bisogna incorporare nel programma alcune informazioni che permettano di caricare (*load*) il programma nella RAM dove la CPU possa acedervi. Inoltre il programma potrebbe essere composta

da più parti che quindi vanno assemblate insieme. Tutti questi compiti sono svolti dal programma chiamato *linker* che trasforma l'object in un programma effettivamente eseguibile. Il programma generato dal linker viene chiamato *eseguibile*. Il programma eseguibile può adesso essere caricato nella RAM dal *loader* ed eseguito.

Il processo di compilazione ed esecuzione di un programma può essere schematizzato come segue:



Sistema Operativo

Il Sistema Operativo di un computer è l'insieme del software necessario al funzionamento di base del computer ed include:

- File System per salvare programmi e dati sulle memorie di massa
- Gestione periferiche come terminali, tastiere, dischi, connessioni, etc.
- Comandi di gestione del sistema operativo
- Applicativi come compilatori, editori di testo, word-processors e così via.

Attualmente i sistemi operativi più diffusi sono UNIX, Linux, MacX e Windows.

2. Linguaggio C

2.1. Introduzione (Rev. 2.1.3)

La nascita del linguaggio C agli inizi degli anni '70 è intimamente legata allo sviluppo del sistema operativo UNIX. La prima versione del sistema operativo UNIX risale al 1970 e fu sviluppata ai Bell Laboratories da Ken Thompson utilizzando un Assembler ed il linguaggio di programmazione B da lui stesso appositamente progettato. Il linguaggio B era basato sul linguaggio BCPL, un linguaggio privo di tipi sviluppato da Martin Richard nel 1967 per la programmazione dei sistemi operativi e che utilizzava come tipi di dati base la parola del processore e faceva un uso pesante dei puntatori e dell'aritmetica degli indirizzi di memoria. Il linguaggio C, sviluppato da Dennis Ritchie su un PDP11 tra il 1968 ed il 1973 ai Bell Laboratories, nasce per sopperire alle limitazioni del linguaggio B e rappresenta quindi un'evoluzione dei linguaggi B e BCPL che incorpora l'utilizzo dei tipi.

Sebbene il linguaggio C nasca come linguaggio per la programmazione del sistema operativo UNIX si distingue dai suoi predecessori per fatto che è a tutti gli effetti un linguaggio di programmazione *general purpose*. Questo ha fatto si che nonostante la sua nascita molto specifica il linguaggio C sia uno dei più diffusi e popolari linguaggi di programmazione general purpose al mondo.

La popolarità del linguaggio C è dovuto ai suoi numerosi vantaggi. Il linguaggio C contiene meno parole chiave di altri linguaggi come ad esempio il Pascal, ciò nonostante ha tipi di dati molto flessibili ed un insieme di operatori che rendono possibile svolgere con poche istruzioni compiti che in altri linguaggi di programmazione ne richiederebbero molte di più. Il linguaggio C fornisce inoltre un ottimo controllo del computer ed è corredato da una ricca collezione di librerie standard di sistema (*run-time standard library*). La programmazione in linguaggio C è intrinsecamente modulare il che rende molto facile sviluppare e mantenere programmi complessi. Inoltre, anche grazie alla presenza di un *preprocessore* che permette di isolare e/o includere parti di programma a seconda del processore, sistema operativo, compilatore, etc., è molto facile *portare* un programma scritto in C da un computer ad un altro. Infine, ma non ultimo, il compilatore C può essere realizzato in modo naturale sull'architettura della maggior parte dei computers.

Naturalmente il linguaggio C non è esente da imperfezioni. La sua sintassi è spesso complicata e fa un uso pesante di simboli come "`*`", "`&`" o "`=`". Un errore tipico di programmazione in C è, ad esempio, l'uso di "`=`" al posto di "`==`" o viceversa. Il C manca inoltre di un controllo automatico sulle dimensioni degli arrays, ed in generale sull'utilizzo della memoria. Infine il linguaggio C non è un linguaggio posizionale per cui non solo le istruzioni possono iniziare ad una colonna qualsiasi e continuare su più righe ma possono esservi più istruzioni sulla stessa riga. Questa libertà permette sia di scrivere programmi facilmente leggibili come programmi totalmente incomprensibili. Tuttavia, nonostante queste imperfezioni, il linguaggio

C è uno dei linguaggi di programmazione più popolari, come testimoniano milioni di linee di programma scritte in C, anche se la sua popolarità è stata di recente un pò offuscata dal suo fratello maggiore il linguaggio C++. Il linguaggio C è inoltre alla base di praticamente tutti i sistemi operativi principali, incluso Windows, e di molti linguaggi di programmazione più evoluti come ad esempio lo stesso C++.

Lo Standard C

Come succede per le lingue parlate anche per il linguaggi di programmazione esistono differenti forme o *dialetti*. Strettamente parlando quello che definisce il dialetto è il compilatore usato. Chiaramente una simile libertà poneva dei limiti alla portabilità ed al mantenimento dei programmi scritti in linguaggio C. Di conseguenza per ovviare a questi problemi a partire dagli inizi degli anni ′80 è iniziato un processo di standardizzazione del linguaggio C che ha portato alla definizione dello *Standard C*.

K&R C e Traditional C

Il linguaggio C fu realizzato da Dennis Ritchie tra gli anni 1969 e 1973, tuttavia la prima definizione formale del linguaggio venne solo nel 1978 con il libro "The C programming language" scritto da Dennis Ritchie e Brian Kernigham. Questa prima definizione del linguaggio è oggi nota come linguaggio "K&R C". Il linguaggio continuò ad evolversi dopo la pubblicazione del libro più o meno liberamente senza degli standards prefissati, inoltre il libro lasciava molti aspetti del linguaggio non ben definiti dando così libertà agli sviluppatori dei compilatori di trattarli come meglio credevano. Il risultato era che il comportamento dei programmi scritti in linguaggio C poteva dipendere dal compilatore o dal sistema su cui venivano compilati. Questa definizione del linguaggio C dei primi anni '80 prima della standardizzazione viene comunemente chiamata "Traditional C". Questo dialetto non è più usato ed è oggi supportato in molti compilatori solo per motivi storici o per poter compilare programmi molto vecchi.

C89 e C95

La standardizzazione del linguaggio C iniziò nel 1982 quando l'American National Standard Institute (ANSI) creò il comitato X3J11 con lo scopo di fornire una definizione standard del linguaggio C e delle sue librerie run-time fissando le parti più oscure del linguaggio. Lo scopo primario era quello di promuovere la compatibilità, l'affidabilità, la facilità di mantenimento ed l'efficienza dei programmi scritti in linguaggio C su una grande varietà di computers. Il comitato terminò i lavori nel 1989 con una definizione del linguaggio molto più precisa e chiara nota come Standard "ANSI C" (X3.159-1989).

Lo standard ANSI C fu ratificato un anno dopo, nel 1990, dal gruppo WG14 dell'International Standardization Organization (ISO) che dopo avervi apportato piccole modifiche per l'internazionalizzazione lo trasformò nello standard internazionale ISO/IEC 9899:1990 o semplicemente 'ISO C". Non vi sono differenze tecniche tra la forma ANSI C e la forma ISO C e questo standard in entrambe le forme è comunemente noto come come "Standard C (1989)" o semplicemente "C89" o a volte, ma più raramente, come "C90". La definizione dello Standard

C (1989) è stata corretta successivamente da due note tecniche di errata (*bug fix*) rilasciate dal comitato WG14 nel 1994 e 1996.

Le modifiche principali introdotte dallo Standard C (1989) rispetto al Traditional C includono oltre ad una chiarificazione di alcune parti del linguaggio come le dichiarazioni e regole di conversione: l'introduzione di una libreria standard, nuove direttive e comandi per il preprocessore, i prototipi delle funzioni, il tipo `void`, nuovi qualificatori come `const` e `signed`. Sono stati inoltre introdotti nuovi tipi di carattere e stringhe per la rappresentazione dei caratteri utilizzati da alcune lingue (*wide* e *multibyte characters*).

Nel 1995 il comitato WG14 fornì una estensione dello Standard C (1989) introducendo piccoli cambiamenti e nuove librerie. Questa definizione del linguaggio è nota come "C89 Amendment 1" o a volte "C94" o "C95". Le modifiche principali includono: nuove macros per la sostituzione degli operatori e caratteri non disponibili nei set di caratteri usati da alcune lingue e tre nuovi file di header standard `wchar.h`, `wctype.h` e `iso646.h`. Sono stati inoltre aggiunte nuove direttive di formattazione per le funzioni di Input/Output `printf()` e `scanf()` e molte nuove funzioni nelle librerie standard.

C99

Lo standard del linguaggio C viene aggiornato periodicamente dal gruppo WG14. L'ultimo aggiornamento è stato rilasciato nel 1999 dopo un lavoro di revisione di tutto il linguaggio iniziato nel 1995. Questa nuova forma ISO/IEC 9899:1999 sostituisce tutte le precedenti ed è al momento la definizione ufficiale dello Standard C, comunemente nota anche come "C99". La definizione è stata corretta con una nota tecnica rilasciata nel 2001.

Il C99 introduce molte nuove estensioni al linguaggio ed alle librerie, tra le più importanti vi sono estensioni per i tipi interi e tipi floating-point, l'aritmetica dei numeri complessi, arrays di dimensione variabile, il tipo Boolean ed i commenti tipo C++ (`//`).

Il C99 rappresenta una grossa evoluzione del linguaggio C rispetto allo standard C89/C95 che però lascia inalterata la natura base del linguaggio.

Dal C al C++

Il linguaggio C è alla base di molti linguaggi di programmazione più avanzati o specialistici come esempio l'Objective-C che permette di utilizzare oggetti, mentre altre varianti di C permettono di sfruttare il parallelismo tra computers e/o processori.

Tra i linguaggi di programmazione più evoluti il C++ occupa un posto di particolare importanza. Nel 1980 Bjarne Stroustrup cominciò a lavorare su una estensione del linguaggio C che includesse le classi, dando origine a quello che oggi è conosciuto come il linguaggio C++, un linguaggio di programmazione fortemente orientato agli oggetti. Grazie ai suoi notevoli miglioramenti rispetto al linguaggio C il linguaggio C++ ha oggi soppiantato il C in molte applicazioni complesse. Il C++ è stato standardizzato nel 1988 ed la sua formulazione standard è nota come "Standard C++".

Lo Standard C++ è per molti aspetti una versione "super" dello Standard C cosicchè non

solo molti compilatori C sono in realtà compilatori C/C++ ma è anche possibile scrivere programmi utilizzando il sottoinsieme comune dello Standard C e C++, noto come "Clean C", che quindi possono essere compilati sia come programmi C che programmi C++.

A questo punto si pone la domanda: è meglio il C o il C++? La risposta a questo quesito dipende dall'interlocutore. La programmazione in C richiede di "partire da zero" dichiarando ad esempio tutti gli oggetti che si vogliono usare. Al contrario il C++ fa molte cose automaticamente senza che il programmatore se ne debba curare. Questo rende la programmazione in C++ più semplice, ma rende al contempo il controllo del programma più difficile poiché richiede di conoscere esattamente quello che il programma stà eseguendo. Di conseguenza alcuni sostengono che il C++ è superiore perché fa molte cose automaticamente ed il C no. Altri invece sostengono che il C è migliore esattamente per lo stesso motivo.

Il nostro punto di vista è che siccome il C lascia quasi tutto il lavoro al programmatore il suo studio fornisce delle conoscenze di base che altri linguaggi non danno, conoscenze che semplificano enormemente l'eventuale apprendimento del linguaggio C++ o di altri linguaggi di programmazione come Perl, Java, Python o altri.

Quale dialetto si usa?

Lo scopo di questo Primer è quello di insegnare la programmazione in linguaggio C senza richiederne una qualche conoscenza precedente, di conseguenza per ridurre al minimo la quantità di informazioni mantenendo al contempo un certo livello di rigore nel linguaggio useremo principalmente il C89 e quanche volta il C89 Amendment 1 trascurando tutte le estensioni introdotte dal C99 ed altre possibili estensioni non standard del linguaggio. Questa scelta, per certi versi limitativa, è fatta per cercare di fornire un dialetto il più possibile compatibile su sistemi diversi ed una base comune alle possibili estensioni di linguaggio. Di conseguenza sebbene lo Standard C ufficiale sia il C99 in questo primer il termine "Standard C" o "Standard ISO C" sarà riferito a quelle parti del C89 comuni anche al C99 indicando esplicitamente, se necessario, le eventuali estensioni introdotte dal C99.

2.2. Programmazione in Linguaggio C (Rev. 2.1.2)

Prima di iniziare la presentazione vera e propria del linguaggio C riassumiamo rapidamente alcuni concetti di base della programmazione in un linguaggio di alto livello, ed in particolare in linguaggio C. Questi verranno poi ripresi ed ampliati più avanti ed in particolare nei vari esempi ed applicazioni discusse.

Un *programma* in linguaggio C è una sequenza di *istruzioni* formulate utilizzando la *sintassi* del linguaggio C che vengono eseguite dal computer sequenzialmente secondo l'ordine con cui sono scritte. Nella programmazione in linguaggio C le istruzioni sono tipicamente raggruppate in *funzioni* che eseguono ben determinate operazioni. Tra tutte le funzioni che compongono un programma in linguaggio C ve ne deve *sempre* essere una, ed una sola, chiamata `main` da cui inizia l'esecuzione del programma.

Le istruzioni che compongono il programma possono essere contenute sia in un unico *file*

ovvero essere distribuite in più *files*. In questo secondo caso il programma è composto da *tutti* i files che contengono le istruzioni del programma. I files con le istruzioni sono chiamati *files sorgente* (*Source File*) ed il loro nome deve terminare con il *suffisso* "`.c`" per indicare che il loro contenuto sono istruzioni in linguaggio C.

Accanto ai files sorgente nella programmazione in linguaggio C esistono i *files* di *header* (*Header File*) che contengono le informazioni che devono essere scambiate tra i vari files sorgente che compongono il programma. Il nome dei files di header termina con il *suffisso* "`.h`" per distinguerli dai files sorgente e sono inclusi in questi ultimi utilizzando la direttiva `#include` del preprocessore.

Un programma in linguaggio C non è direttamente utilizzabile dal computer in quanto il processore non comprende le istruzioni del linguaggio C per cui il programma deve essere "tradotto" mediante il *compilatore* C nelle istruzioni del *linguaggio macchina* specifico del processore. Il compilatore, a differenza del processore, "capisce" il linguaggio C per cui analizza il file sorgente e se non vi sono errori di sintassi produce un file che contiene la traduzione in linguaggio macchina delle istruzioni in linguaggio C contenute nel file sorgente. Il file prodotto dal compilatore viene chiamato usualmente *codice oggetto* (*Object Code*) ed il file che lo contiene ha di norma lo stesso nome del file sorgente con il suffisso "`.c`" sostituito dal suffisso "`.o`" per indicare che il file contiene un codice oggetto. Se il programma è composto da più files sorgente il compilatore analizza separatamente ciascun file sorgente producendo per ciascuno di essi il corrispondente file con il codice oggetto.

Quando tutti i files sorgente sono stati compilati i files con il codice oggetto sono passati al programma chiamato *linker* che ha il compito di assemblare insieme le varie parti del programma controllando che tutte le funzioni utilizzate nel programma siano state incluse ed aggiungendo, se necessario, funzioni dalle librerie *run-time* standard. Se non vi sono errori il linker produce un unico file con il programma in linguaggio macchina che può ora essere eseguito dal computer.

Il *linker* non è specifico del linguaggio C, infatti ogni computer ha un suo linker di sistema che può essere usato per produrre i programmi eseguibili a partire da codici oggetto generati da files sorgenti scritti anche con linguaggi di programmazione differenti dal C. Con opportuni accorgimenti il linker può anche produrre programmi eseguibili assemblando codici oggetto prodotti da files sorgente scritti con linguaggi di programmazione differenti tra loro.

Sebbene tutti i computers utilizzino la procedura appena descritta, ma non essendo le cose mai semplici come potrebbero essere, i dettagli della traduzione di un programma in linguaggio C nel corrispondente programma eseguibile dipendono sia dal sistema operativo utilizzato che dal compilatore. Inoltre se si utilizzano ambienti di sviluppo come ad esempio KDevelop l'intero processo può risultare completamente nascosto. Per i dettagli specifici rimandiamo quindi alla documentazione del computer utilizzato, qui ci limiteremo a riportare in Appendice la procedura per un sistema UNIX che utilizzi il compilatore GNU C.

2.2.1. Primo Programma in Linguaggio C

Per illustrare gli elementi base e la struttura di un programma in linguaggio C consideriamo il seguente semplice programma contenuto nel file sorgente `hello1.c` che stampa sul terminale

la frase: "Hello from your C program".

Programma: `hello1.c`

```
#include <stdio.h>

int main(void)
{
  printf("Hello from your C program\n");
  return 0;
}
```

Nonostante la sua semplicità questo programma contiene molte delle caratteristiche principali dei programmi scritti in C.

- `#include <stdio.h>`

  Questa non è un'istruzione del linguaggio C ma una *direttiva* per il preprocessore. Tutte le direttive per il preprocessore iniziano con il carattere "`#`". In questo caso la direttiva è quella di includere nel file sorgente `hello1.c` il contenuto del file di header `stdio.h` che contiene le informazioni necessarie al programma per utilizzare la funzione delle librerie standard di sistema `printf()`. Il file di header `stdio.h` è un file delle librerie standard la cui collocazione nel file system dipende in generale sia dal sistema operativo che dal compilatore utilizzati. Per ovviare all'inconveniente di dover sapere esattamente dove si trova il file si usano i caratteri "`<`" e "`>`" per indicare al preprocessore di cercare il file nel posto dove sono conservati i file di sistema. In questo modo il programmatore non deve curarsi delle specifiche del sistema operativo ed il programma risulta al contempo più portabile.

- `int main(void)`

  Ogni programma in C contiene una funzione chiamata `main()` da cui inizia l'esecuzione del programma. Questa riga letteralmente significa: "`main()` è una funzione priva di parametri che ritorna un valore di tipo `int` (intero)".

  Le parentesi dopo l'identificatore `main` sono *necessarie* anche in assenza di parametri perché indicano che `main` è una funzione.

- {
  ...
  }

  Le parentesi graffe "{}" sono usate per raggruppare più istruzioni, in questo caso racchiudono il corpo della funzione `main()`, ossia l'insieme delle istruzioni che definiscono l'operato della funzione.

- `printf("Hello from your C program\n");`

  Nel linguaggio C molte operazioni, anche elementari, non vengono fatte con istruzioni primitive del linguaggio ma utilizzando funzioni che si trovano nelle librerie *run-time*

standard di sistema. Queste sono una raccolta di funzioni raggruppate per tipologia che formano una parte essenziale del compilatore. In questo caso per stampare un messaggio sul terminale si utilizza la funzione `printf()` che fa parte della libreria standard di Input/Output.

Il nostro semplice programma è quindi composto da almeno due files sorgente: il file `hello1.c` ed il file sorgente delle librerie standard che contiene la definizione della funzione `printf()`. Le parti del programma si scambiano informazioni attraverso il file di header `stdio.h` che contiene le informazioni necessarie per utilizzare le funzioni della libreria standard di Input/Output.

Il programmatore non si deve curare di compilare la parte relativa alle librerie standard in quanto i loro codici oggetto sono già disponibili nel sistema organizzati in librerie. È poi compito del linker assemblare insieme i vari codici oggetto necessari utilizzando le librerie opportune.

- `"Hello from your C program\n"`

Questa è una costante di tipo stringa e forma l'argomento o parametro della funzione `printf()`. In C una sequenza di caratteri delimitata da una coppia di doppi apici "`"`" è chiamata stringa. La stringa contiene sia i caratteri da stampare che le informazioni sulla formattazione del testo sul terminale. Lo spazio bianco (*space*) è un carattere, usualmente indicato con "␣", per cui la stringa da stampare è composta dai seguenti caratteri:

```
"Hello␣from␣your␣C␣program"
```

Nel seguito, a meno che non sia necessario, il carattere *space* non sarà indicato esplicitamente nelle stringhe. L'ultima parte della stringa "`\n`" (*newline*) è un carattere speciale di *escape* per la formattazione del testo sul terminale formato dal carattere *backslash* "`\`" e dal carattere"`n`" che indica di andare a capo una volta scritti tutti i caratteri della stringa.

- `...;`

Il linguaggio C è non è un linguaggio posizionale: le istruzioni possono iniziare in qualsiasi colonna e continuare su più righe. Il carattere "`;`" (*semicolon*) indica la fine di una istruzione. Più istruzioni possono quindi essere scritte sulla stessa riga, purché separate dal separatore "`;`".

Questa proprietà unita la fatto che i caratteri come lo spazio bianco o il *tab* sono generalmente ignorati nei file sorgenti, a meno che non compaiano ad esempio nelle stringhe, permette di scrivere i programmi in linguaggio C in una forma facilmente leggibile ovvero, se usata male, in una forma completamente incomprensibile. È una buona norma di programmazione quella di cercare di scrivre i programmi nel modo più chiaro possibile.

- `return 0;`

Questa istruzione indica che l'esecuzione della funzione `main()` deve essere terminata. Quando si incontra questa istruzione l'esecuzione della funzione termina restituendo alla

parte di programma che ha chiamato la funzione, o in questo caso al sistema operativo, come valore della funzione il valore che segue l'istruzione `return`. Nel nostro esempio la funzione `main()` ha quindi il valore `0`. Sebbene il compilatore accetti di compilare funzioni senza l'istruzione `return` è buona norma terminare ogni funzione con un'istruzione di `return`.

Per eseguire il programma bisogna compilarlo. Come detto in precedenza la compilazione dipende in generale dal sistema operativo e dal compilatore utilizzati. Nel seguito supporremo sempre di trovarci in ambiente UNIX ed di utilizzare un compilatore GNU C. Sotto queste ipotesi per compilare eseguire il programma basta eseguire i comandi:

```
$ cc hello1.c
$ a.out
Hello from your C program
$
```

Osserviamo che il *prompt* "$" della linea di comando si trova sulla linea successiva a quella su cui è stata scritta la stringa poichè l'ultimo carattere della stringa è il *newline*. Se si fosse omesso "\n" il prompt sarebbe apparso attaccato alla "m" di "program".

Alcune semplici osservazioni sulla funzione `printf()`

La funzione `printf()` scrive la stringa di caratteri sul terminale da sinistra a destra partendo dalla posizione successiva all'ultima posizione di scrittura utilizzata. Questo vuol dire che se la funzione è chiamata più volte ciascuna chiamata inizia a scrivere nella posizione immediatamente successiva a quella in cui ha finito di scrivere la chiamata precedente. Il seguente programma produce quindi lo stesso risultato di `hello1.c`:

Programma: `hello2.c`

```c
#include <stdio.h>

int main(void)
{
  printf("Hello ");
  printf("from your ");
  printf("C program\n");
  return 0;
}
```

Di conseguenza per andare a capo non basta chiamare separatamente la funzione `printf()` ma si deve inserire esplicitamente nella stringa il carattere di *newline* "\n" nel punto voluto. Così se ad esempio volessimo scrivere la frase "Hello from your C program" su due linee differenti è necessario utilizzare due "\n" come mostrato dal programme seguente:

Programma: `hello3.c`

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

```
int main(void)
{
  printf("Hello \n");
  printf("from your C program\n");
  return 0;
}
```

che produce sullo schermo la scritta:

```
Hello
from your C program
```

Non è necessario usare una funzione `printf()` per ogni riga di stampa poichè il carattere di *newline* può essere messo in ogni punto della stringa per cui lo stesso risultato di `hello3.c` si può ottenere anche come

Programma: `hello4.c`

```
#include <stdio.h>

int main(void)
{
  printf("Hello \nfrom your C program\n");
  return 0;
}
```

Questa proprietà della funzione `printf()` premette di formattare facilmente l'output, come mostra il seguente programma:

Programma: `hello5.c`

```
#include <stdio.h>

int main(void)
{
  printf("\n\n\n\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣**********\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣Hello␣from␣␣␣␣␣␣␣␣␣*\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*␣your␣C␣program␣␣␣␣␣␣␣*\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣**********\n");
  printf("\n\n\n\n");
  return 0;
}
```

che quando eseguito produce sullo schermo la scritta:

```
**********
* Hello from           *
* your C program       *
**********
```

Avremo molto di da dire più avanti sulla formattazione dell'output.

2.3. Elementi Lessicali (Rev. 2.1.2)

Un file sorgente contenente istruzioni in linguaggio C è composto da una sequenza di caratteri che opportunamente divisi in gruppi formano le unità lessicali del linguaggio, chiamate anche *tokens*. Un file sorgente non differisce quindi troppo da un libro. Anche questo infatti a ben vedere non è altro che una sequenza di caratteri che opportunamente suddivisi e raggruppati formano le parole, ossia gli elementi lessicali, della lingua in cui è scritto il libro. La trasformazione delle sequenze di caratteri in unità lessicali è chiamata analisi lessicale o in inglese *tokenization*. Come le unità lessicali del libro sono poi raggruppate per formare le frasi, così allo stesso modo le unità lessicali del linguaggio C sono raggruppate tra loro per formare le istruzioni del linguaggio.

2.3.1. Caratteri

Le unità lessicali del linguaggio C sono formate utilizzando i seguenti caratteri:

- i 52 caratteri alfabetici Latini maiuscoli e minuscoli:

  ```
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  ```

- i 10 digits decimali:

  ```
  0 1 2 3 4 5 6 7 8 9
  ```

- Lo spazio bianco "*space*" ed il tab "*tab*"

- i seguenti caratteri grafici:

!	p. esclamativo	#	numero	%	percento
^	acc. circonflesso	&	ampersand	*	asterisco
(	par. tonda sx	)	par. tonda dx	-	meno
_	underscore	+	più	=	uguale
~	tilde	[	par. quadra sx	]	par. quadra dx
\	backslash	|	barra verticale	;	punto e virgola
:	due punti	'	apostrofo	"	double quote
{	par. graffa sx	}	par. graffa dx	,	virgola
.	punto	<	minore	>	maggiore
/	slash	?	p. interrogativo		

Trigraph

I set di caratteri utilizzati dalle diverse lingue non sono necessariamente tutti uguali così può accadere che non tutti i caratteri siano sempre disponibili, in particolare possono mancare alcuni dei 29 caratteri grafici. Ad esempio sulle tastiere italiane mancano le parentesi graffe "{" "}".

In genere i caratteri mancanti possono essere introdotti con opportune sequenze di escape, tuttavia per favorire l'internazionalizzazione del linguaggio nello Standard C89 Amendment 1 sono stati introdotti 9 trigrammi (*trigraph*), ossia gruppi di tre caratteri che formano un solo fonema, per la sostituzione dei caratteri che mancano con più frequenza. Questo permette di scrivere i programmi in linguaggio C utilizzando solo un insieme ristretto di caratteri noto come ISO 646-1083 Invariant Code Set formato da un insieme di caratteri presenti in praticamente tutti gli insiemi di caratteri utilizzati dalle varie lingue.

Trigraph	Sostituisce	Trigraph	Sostituisce
??(	[	??)	]
??<	{	??>	}
??/	\	??!	|
??'	^	??-	~
??=	#		

La sostituzione dei *trigraphs* con i corrispondenti caratteri avviene prima dell'analisi lessicale del file sorgente, questo vuol dire che tutte le occorrenze di triplette di caratteri interpretabili come uno dei nove trigraph sono *sostituite* con il corrispondente carattere prima di procedere alla divisione dei caratteri in unità lessicali. Di conseguenza se nel nostro programma hello1.c avessimo scritto

Programma: `hello.c`

```
#include <stdio.h>

int main(void)
{
  printf("??(Hello from your C program??)\n");
  return 0;
}
```

sul terminale verrebbe scritto

```
[Hello from your C program]
```

Per evitare che una sequenza di tre caratteri sia interpretata come un trigraph si deve utilizzare il carattere di escape "\?" al posto del carattere "?" per cui se la stringa viene scritta come

```
"\?\?(Hello from your C program\?\?)\n"
```

adesso il programma scriverà sullo schermo

```
??(Hello from your C program??)
```

Operator Macros

Lo Standard C89 Amendment 1 introduce oltre ai trigraphs anche il file di header `iso646.h` che contiene la definizioni delle seguenti *macros*

Macro	Operatore	Macro	Operatore
`and`	`&&`	`not_eq`	`!=`
`and_eq`	`&=`	`or`	`||`
`bitand`	`&`	`or_eq`	`|=`
`bitor`	`|`	`xor`	`^`
`compl`	`~`	`xor_eq`	`^=`
`not`	`!`		

utilizzabili in sostituzione dei corrispondenti operatori. Le macros sono sostituite con il corrispondente operatore dal preprocessore prima di iniziare l'analisi lessicale.

2.3.2. Identificatori

Gli identificatori servono per dare nomi unici ai vari oggetti del programma. Un identificatore è un *token* composto da una sequenza di caratteri alfabetici maiuscoli o minuscoli, digits decimali e *underscore* "_". Un identificatore non può iniziare con un digit e non deve coincidere con una parola chiave (*keywords*) del linguaggio.

Esempi:

```
k
_ind
identificatore
numero_dati_1

3_dati              /* errato: non puo' cominciare con una cifra */
3input              /* errato: non puo' cominciare con una cifra */
numero%dati         /* errato: il carattere % non e' permesso    */
numero dati         /* errato: contiene uno spazio               */
int                 /* errato: e' una parola chiave              */
```

Due identificatori sono uguali se sono scritti scritti allo stesso modo, incluso il caso dei caratteri alfabetici, per cui `abc`, `Abc` o `ABC` sono identificatori diversi. I compilatori più vecchi consideravano due identificatori uguali se i coincidevano i primi 8 caratteri, indipendentemente dai restanti. Ad esempio gli identificatori `i_am_an_identifier` e `i_am_an_elephant`

venivano considerati uguali. Lo standard C89 richiede che almeno i primi 31 caratteri di un identificatore siano rilevanti, tuttavia molti compilatori C ne usano anche di più. Lo standard C99 ha elevato questo limite a 63 caratteri.

Un'altra cosa a cui bisogna stare attenti, oltre a non usare per identificatori le parole chiave del C, è di non utilizzare nomi usati nelle librerie standard. Abbiamo visto ad esempio che per scrivere una stringa sul terminale si usa la funzione `printf()` definita nelle librerie standard, di conseguenza il token `printf` non deve essere utilizzato come identificatore essendo già usato per una funzione delle librerie standard.

Bisogna anche evitare identificatori che iniziano con due *underscore* "__" o con un *underscore* "_" seguito da una lettera maiscola perché lo Standard C riserva questi identificatori per scopi di sistema. Ad esempio nel file di header `stdio.h` delle librerie standard vengono usati identificatori come `_G_fpos_t` o `__io_funcs`.

Per aumentare la chiarezza di un programma è buon stile di programmazione utilizzare come identificatori nomi significativi, con eventualmente uno o più underscore, che rendano la comprensione delle istruzioni più immediato. Ad esempio l'identificatore `numero_di_dati` è sicuramente più facilmente leggibile di `numerodidati`, così come il significato dell'istruzione

```
media = somma / numero_di_dati;
```

è sicuramente più chiaro di quello di

```
x = y / z;
```

Alcuni compilatori permettono di utilizzare negli identificatori il segno del dollaro "$" per scopi particolari come accedere a funzioni non standard-C fornite da alcuni sistemi operativi.

2.3.3. Parole Chiave

Le parole chieve o *keywords* sono identificatori a cui il linguaggio C attribuisce un significato ben preciso e che quindi non possono essere utilizzate come identificatori in contesti diversi dalla loro definizione. Una parola riservata può essere ridefinita utilizzando il preprocessore, questa però è considerata una cattiva pratica poiché rende il programma meno chiaro.

Le keywords del C89 sono:

`auto`	`do`	`goto`	`signed`	`unsigned`
`break`	`double`	`if`	`sizeof`	`void`
`case`	`else`	`int`	`static`	`volatile`
`char`	`enum`	`long`	`struct`	`while`
`const`	`extern`	`register`	`switch`	
`continue`	`float`	`return`	`typedef`	
`default`	`for`	`short`	`union`	

Il C99 introduce le keywords aggiuntive: `inline`, `restrict`, `_Bool`, `_Complex` e `_Imaginary`.

Oltre alle keywords del linguaggio è buona norma considerare come riservati anche i nomi

delle *macros* `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor` and `xor_eq` definite nel file di header `iso646.h` delle librerie standard.

2.3.4. Operatori e Separatori

Gli operatori indicano le operazioni da effettuarsi con gli oggetti su cui operano, mentre i separatori sono i caratteri di interpunzione del linguaggio. Gli operatori ed i separatori usati dal C possono essere divisi nelle seguenti classi lessicali:

Classe Lessicale	**Tokens**
Operatori semplici	`! % ^ * - + = ~ | . < > / ?`
Operatori assegnamento composti	`+= -= *= /= %= <<= >>= &= ^= |=`
Altri operatori composti	`-> ++ -- << >> <= >= == != && ||`
Separatori	`( ) [ ] { } , ; : ...`

Nello Standard C gli operatori di assegnamento composti formano un unico token per cui non vi possono essere spazi bianchi tra l'operatore ed il segno di uguale "`=`".

Tra i separatori vanno inclusi anche lo *space* "␣" ed il *tab* "⟶". Il separatore punto e virgola "`;`" indica la fine di un'istruzione. Piú genericamente si chiama *istruzione* una sequenza di tokens terminata dal carattere "`;`".

Il significato di alcuni operatori e/o separatori può dipendere dal contesto. Ad esempio il significato delle parentesi tonde "`()`" nel seguente pezzo di programma

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a, b, c;
6    a = 2 * ( b + c );
7    ......
```

dipende chiaramente dal contesto. Alla linea 3 le parentesi subito dopo l'identificatore `main` segnalano al compilatore che `main()` è una funzione, mentre alla linea 6 indicano al compilatore l'ordine con cui devono essere valuatate le varie parti dell'espressione: prima la somma `b+c` e poi risultato moltiplicato per 2. In questi casi si parla di *overloading* dell'operatore o del separatore.

Infine ricordiamo che per semplificare la scrittura dei programmi nel caso non si abbiano a disposizione alcuni caratteri lo Standard C89 Amendment 1 ed il C99 utilizzano per i separatori "`{`", "`}`", "`[`", "`]`" ed il carattere "`#`" rispettivamente la scrittura alternativa "`<%`", "`%>`", "`<:`", "`:>`" e "`%:`" in aggiunta ai trigraphs.

2.3.5. Commenti

I commenti vengono utilizzati come supporto nella scrittura e nella documentazione di un programma per spiegarne il funzionamento e l'utilizzo. L'uso dei commenti semplifica sia lo sviluppo che l'eventuale modifica dei programmi per questo è buona norma scrivere i commenti durante la scrittura del programma.

I commenti in C sono stringhe di simboli di lunghezza arbitraria racchiuse dai delimitatori "/*" e "*/". Questo vuol dire che un commento inizia con l'occorrenza dei due caratteri "/*" e finisce con la prima occorrenza dei due caratteri "*/" e pertanto commenti all'interno di commenti non sono riconosciuti. I commenti vengono sostituiti dal compilatore con un singolo spazio bianco, e pertanto non fanno parte del programma eseguibile.

Esempi:

```
/* Questo e' un commento */

/* --- Questo e' un altro commento --- */

/*   Un commento puo' essere scritto anche su
 *   piu' line. In questo modo e' possibile
 *   inserire piu' informazioni
 */

/*************************************************
 *    Un programma deve scritto in modo ordinato    *
 *    e leggibile. Per questo e' buona norma        *
 *    evidenziare i commenti. Ad esempio come       *
 *    questo.                                       *
 *************************************************/
```

Con il C99 i commenti possono iniziare anche con i caratteri "//" ed estendersi, in questo caso, fino alla fine della riga.

Esempio:

```
// Questo e' un commento
// permesso dal C99 ma non dal C89
```

I commenti sono sostituiti dal compilatore con un spazio bianco prima del passaggio del file al preprocessore, di conseguenza gli eventuali comandi e direttive per il processore che si trovino all'interno dei commenti non saranno eseguite.

2.4. Tipi base (Rev. 2.1.4)

Un *tipo* è definito da un insieme di valori corredato dall'insieme di operazioni che possono essere effettuate con essi. Ad esempio per un tipo intero i valori sono gli interi in un determinato intervallo e le operazioni sono la somma, la sottrazione, la moltiplicazione, la divisione,

le disuguaglianze ed uguaglianze e così via. Analogamente un tipo floating-point prenderà valori rappresentabili come numeri a virgola mobile, ad esempio decimali, e sarà corredato dall'insieme di operazioni associate. È importante osservare che sebbene tipi diversi possano possedere operazioni simili, ad esempio la somma o la divisione per tipi interi e floating-point, la definizione delle operazioni e quindi il risultato delle stesse può dipendere dal tipo. Avremo modo di fare qualche esempio più avanti.

Dire che un oggetto, ad esempio una variabile od una espressione, è di "tipo `T`" significa che i suoi valori sono ristretti al dominio di variazione di `T` e che con questo possono essere effettuate tutte le operazioni associate a `T`.

In un computer digitale tutte le informazioni sono rappresentate da sequenze di bits, di conseguenza affinché un tipo possa essere manipolato dal computer è necessario *codificarlo* come sequenze di bits. Il numero di bits utilizzati nella codifica di un tipo si chiamata *dimensione* o *lunghezza* del tipo. La dimensione e la codifica utilizzata, così come i valori che il tipo può prendere, in generale dipendono dal sistema.

Il linguaggio C fornisce molti tipi primitivi, come ad esempio vari tipi di interi e numeri floating-points, puntatori, funzioni, strutture e così via. Tutti questi tipi verranno considerati via via in questo primer iniziando qui dai tipi utilizzati per rappresentare i Numeri interi, i Caratteri ed i Numeri reali.

2.4.1. Tipo Intero

Il linguaggio C fornisce più tipi interi ed operatori ad essi associati di molti altri linguaggi di programmazione. I tipi interi differiscono tra loro sia nella rappresentazione, numero di bits e codifica utilizzati, che nell'insieme delle operazioni e dei valori che possono assumere. Il linguaggio C nasce come linguaggio di programmazione di sistemi operativi e questa grande varietà riflette la necessità di doversi adattare alle differenti caratteristiche dei computers come la lunghezza della parola e gli operatori matematici. Il risultato è che il linguaggio C permette una buona corrispondenza dei tipi con le specifiche del processore.

Il tipo intero viene utilizzato nel linguaggio C per rappresentare:

- numeri interi con o senza segno con le usuali operazioni aritmetiche e di relazione.
- sequenze di bits con le operazioni logiche `NOT`, `AND`, etc.
- valori booleani per i quali il valore zero è considerato *falso* (`false`) mentre ogni valore non nullo è considerato *vero* (`true`).
- caratteri rappresentati dal valore numerico della loro codifica.

Lo Standard C richiede che tutti gli interi siano codificati utilizzando una codifica binaria, ossia come sequenze di digits 0 e 1. La codifica utilizzata differisce per interi senza segno ed interi con segno.

Tipo intero senza segno: `unsigned`

Per rappresentare i valori interi senza segno il C89 fornisce tre tipi che differiscono tra loro per il numero di bits utilizzati nella codifica. Partendo dal tipo di dimensione inferiore i tre tipi sono

Tipo intero senza segno:

`unsigned short int`	o	`unsigned short`
`unsigned int`	o	`unsigned`
`unsigned long int`	o	`unsigned long`

Per ogni tipo sono stati indicati i modi equivalenti di specificarlo.

Il C99 introduce il quarto tipo `unsigned long long int` di dimensione ancora più grande.

La *keyword* `unsigned`, caratteristica di questa classe, indica che il tipo è senza segno e quindi specifica sia il tipo di rappresentazione utilizzata che l'insieme delle operazioni associate. Tutti i tipi interi senza segno usano la rappresentazione binaria o in *base* 2.

La rappresentazione binaria, al pari della rappresentazione decimale (o in *base* 10) che usa 10 simboli, è una rappresentazione *posizionale* in cui il valore di ciascun simbolo 0 e 1 dipende dalla sua posizione nella stringa che rappresenta il numero. Ad esempio come 7435 nella rappresentazione decimale *significa*

$$7435_{10} = 7 \times 10^3 + 4 \times 10^2 + 3 \times 10^1 + 5 \times 10^0$$

così 1101 nella rappresentazione binaria significa

$$1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Per trasformare un numero in rappresentazione binaria nell'equivalente in rappresentazione decimale basta sostituire alle potenze di 2 il loro valore in rappresentazione decimale ed effettuare la somma. Nel nostro esempio precedente si ha quindi

$$1101_2 = 13_{10}$$

Per trasformare invece un numero dalla rappresentazione decimale a quella binaria basta dividerlo successivamente per 2 scrivendo da destra a sinistra il resto della divisione, come mostrato qui di seguito per il numero 25:

$$\begin{array}{rcrcr} 25/2 & = & 12 + \text{Resto } 1 & \Rightarrow & 1 \\ 12/2 & = & 6 + \text{Resto } 0 & \Rightarrow & 0 \\ 6/2 & = & 3 + \text{Resto } 0 & \Rightarrow & 0 \\ 3/2 & = & 1 + \text{Resto } 1 & \Rightarrow & 1 \\ 1/2 & = & 0 + \text{Resto } 1 & \Rightarrow & 1 \\ \hline & & & \Rightarrow 25_{10} & = 11001_2 \end{array}$$

Ulteriori informazioni sui sistemi di numerazione digitale più usati possono essere trovate nell'Appendice.

I numeri interi positivi rappresentabili dipendono dal numero di bits utilizzati. Con n bits si possono rappresentare gli interi positivi da 0 a $2^n - 1$ per cui, ad esempio, con 4 bits si possono rappresentare tutti i numeri interi positivi da 0 a 15:

0000_2	=	0_{10}	0110_2	=	6_{10}	1100_2	=	12_{10}
0001_2	=	1_{10}	0111_2	=	7_{10}	1101_2	=	13_{10}
0010_2	=	2_{10}	1000_2	=	8_{10}	1110_2	=	14_{10}
0011_2	=	3_{10}	1001_2	=	9_{10}	1111_2	=	15_{10}
0100_2	=	4_{10}	1010_2	=	10_{10}			
0101_2	=	5_{10}	1011_2	=	11_{10}			

Tipo intero con segno: `signed`

Per ogni tipo senza segno vi è un corrispondente tipo con segno che utilizza lo stesso numero di bits ma un differente metodo di codifica. Come nel caso degli interi senza segno vi sono più modi equivalenti per indicare un tipo intero con segno. Partendo da quello di dimensione più piccola, ed indicando su ogni riga i modi equivalenti, questi sono

Tipo intero con segno:

`short`	o	`short int`	o	`signed short`	o	`signed short int`	
`int`	o	`signed int`	o	`signed`			
`long`	o	`long int`	o	`signed long`	o	`signed long int`	

Come nel caso degli interi senza segno il C99 introduce un quarto tipo dimensione ancora più grande `signed long long int`.

La keyword `signed` è stata introdotta nel C89 e può essere omessa per compatibilità con il Traditional C.

L'intervallo di valori che un tipo intero con segno può assumere dipende oltre che dal numero di bits usati per rappresentarlo anche dal metodo di codifica usato. Il metodo di codifica più usato è la codifica in *complemento a due* in cui il valore di un intero con segno è rappresentato da n bits utilizzando il bit più a sinistra (*most significative bit*) come bit del segno ed i restanti $n - 1$ bits per la codifica del modulo del numero. Il bit del segno vale 0 per i numeri positivi e 1 per quelli negativi.

La codifica del valore del modulo differisce a seconda che il valore sia positivo o negativo. Per i valori positivi si utilizza la normale rappresentazione binaria con $n - 1$ bits. Per i valori negativi si utilizza la codifica ottenuta negando, ossia invertendo

$$0 \Rightarrow 1$$

$$1 \Rightarrow 0$$

ciascun bit della codifica del corrispondente valore positivo ed aggiungendo al risultato 1. Ad esempio

k	Rapp. k	NOT k		Rapp. −k	−k
1	0|0000001	1|1111110	+1 ⇒	1|1111111	−1
9	0|0001001	1|1110110	+1 ⇒	1|1110111	−9
0	0|0000000	1|1111111	+1 ⇒	0|0000000	0
−5	1|1111011	0|0000100	+1 ⇒	0|0000101	5

Per motivi di spazio si è assunta una rappresentazione con 8 bits = byte ed inoltre per chiarezza è stato evidenziato il bit del segno della rappresentazione. Utilizzando le regole di somma binaria

$$\begin{aligned} 0+0 &= 0 \\ 0+1 &= 1 \\ 1+1 &= 10 \end{aligned}$$

è facile verificare che la somma di k e $-k$ in rappresentazione in complemento a due con 8 bits è uguale a zero.

Per comprendere l'origine di questa rappresentazione basta osservare che sommando un bit con il suo complemento il risultato è sempre uguale a 1 qualsiasi sia il valore del bit. Da questo segue che se i numeri sono rappresentati con n bits il risultato della somma del numero k con la sua negazione **NOT** k è una stringa n bits uguali ad 1 ed è quindi uguale alla rappresentazione del numero intero positivo $2^n - 1$. La rappresentazione in complemento a due di $-k$ è data da $1 +$ **NOT** k per cui in questa rappresentazione si ha sempre $k + (-k) = 2^n$, dove n è il numero di bits usati dalla rappresentazione. La prima conseguenza di questa relazione è che la somma $k + (-k)$ in rappresentazione in complemento a due con n bits è sempre uguale a zero. Infatti la rappresentazione binaria di 2^n è data n bits uguali a 0 più uno, l'$n+1$-esimo, uguale a 1 ma quest'ultimo manca e quindi restano solo i primi n bits uguali a 0. Da questa relazione segue inoltre che la rappresentazione in complemento a due con n bits del numero $-k$ è uguale alla rappresentazione binaria del numero positivo $2^n - k$.

Uno dei vantaggi della rappresentazione in complemento a due è che lo stesso hardware che esegue l'addizione può essere utilizzato per eseguire le sottrazioni poiché l'operazione $n - m$ è equivalente a $n + (-m)$, ed $-m$ si ottiene aggiungendo 1 alla negazione di m.

Con la rappresentazione in complemento a due con n bits si possono rappresentare i valori interi da -2^{n-1} a $2^{n-1} - 1$ inclusi, per cui ad esempio con 32 bits (4 bytes) si hanno i valori

$$-2^{31},\ -2^{31}+1,\ \ldots\ 2^{31}-2,\ 2^{31}-1$$

Un'altra codifica utilizzata è la codifica con il bit del segno o *true magnitude* o *sign magnitude* in cui i valori negativi sono codificati negando il solo bit del segno della rappresentazione binaria del corrispondente valore positivo. Nella codifica *true magnitude* i valori positivi e negativi differiscono quindi solo per il valore del bit del segno che vale 0 per i primi e 1 per i secondi, mentre gli altri $n - 1$ bits contengono in entrambi i casi la rappresentazione binaria del modulo del valore.

Esiste anche una codifica in *complemento a uno* in cui i valori negativi sono codificati negando tutti i bits della rappresentazione binaria del corrispondente valore positivo. La codifica in

complemento a uno differisce quindi dalla codifica in complemento a due per l'aggiunta del valore 1.

Sia la codifica in true magnitude che la codifica in complemento a uno permettono di rappresentare con n bits i valori interi da $-(2^{n-1}-1)$ a $2^{n-1}-1$. Si ha un valore in meno rispetto alla codifica in complemento a due ma in compenso si hanno due rappresentazioni dello 0, uno positivo ed uno negativo.

Tutte e tre le codifiche possono essere utilizzate con lo Standard C. Osserviamo tuttavia che le tre codifiche non sono equivalenti, infatti "11111011" rappresenta il valore "−5" in complemento a due e "−123" in true magnitude e "−4" in complemento a uno.

La caratteristica `signed` o `unsigned` di un tipo intero influenza non solo i valori che il tipo può prendere ma anche le operazioni che vengono effettuate su di esso. Ad esempio tutte le operazioni aritmetiche su di un tipo intero `unsigned` seguono l'aritmetica modulo 2^n per cui se al valore massimo di un tipo intero si aggiunge 1 si otterrà sempre il valore 0.

Lo Standard C non fissa le dimensioni dei vari tipi interi ma richiede sia per i `signed` che per gli `unsigned` che la dimensione del tipo `short` sia di almeno 16 bits = 2 bytes, quella del tipo `long` di almeno 32 bits = 4 bytes e che la dimensione del tipo `int` non sia minore di quella del tipo `short` né maggiore di quella del tipo `long`. Su molti sistemi la dimensione del tipo `int` è uguale a quella del tipo `long`.

Il C99 richiede inoltre che il tipo `long long` sia di almeno 64 bits = 8 bytes e che quindi sul sistema siano definite le tutte le operazioni aritmetiche a 64 bits.

Lo Standard C richiede inoltre che la dimensione e le altre caratteristiche dei tipi interi siano definite nel file di header `limits.h` attraverso delle *macros* alcune delle quali sono:

Nome	**Descrizione**	**Valore minimo**
`SHRT_MIN`	valore minimo `signed short int`	−32767
`SHRT_MAX`	valore massimo `signed short int`	32767
`USHRT_MAX`	valore massimo `unsigned short int`	65535
`INT_MIN`	valore minimo `signed int`	−32767
`INT_MAX`	valore massimo `signed int`	32767
`UINT_MAX`	valore massimo `unsigned int`	65535
`LONG_MIN`	valore minimo `signed long int`	−2147483647
`LONG_MAX`	valore massimo `signed long int`	2147483647
`ULONG_MAX`	valore massimo `unsigned long int`	4294967295

Il valore riportato in questa tabella corrisponde al valore minimo richiesto dallo Standard C. Le caratteristiche dei nuovi tipi interi introdotti dal C99 sono definite nel file di header `stdint.h` introdotto anch'esso dallo Standard C99.

2.4.2. Tipo Carattere

Per rappresentare i caratteri il C fornisce i tre tipi:

Tipo carattere:

```
char
signed char
unsigned char
```

Tutti e tre usano lo stesso numero di bits ma differiscono per i valori che possono prendere. In C il tipo utilizzato per i caratteri è un tipo intero per cui i suoi valori sono interi. Ogni carattere viene codificato con una sequenza di bits che interpretati come rappresentazione di un intero forniscono il valore del carattere. È vero anche il contrario ossia la rappresentazione di un intero può essere interpretata come quella di un carattere. Si così una corrispondenza uno a uno tra caratteri ed interi.

Chiaramente a seconda della rappresentazione in bits utilizzata per i caratteri si avranno differenti associazioni tra caratteri e valori interi. La tavola che ad ogni carattere associa la sequenza di bits della rappresentazione, e quindi un valore numerico intero, si chiama codice della rappresentazione.

Esistono molti codici che differiscono sia per il numero di bits utilizzati che per l'associazione tra caratteri e sequenze di bits. Uno dei codici più usati è il codice ASCII a 7-bit:

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

Questa codifica significa che ad esempio il carattere “a” è codificato come il numero intero “97”. Osserviamo che il valore numerico dei digits decimali intesi come caratteri non corrisponde al valore del digit, ad esempio il valore numerico del *carattere* “2” è “50 e non 2.

Il tipo carattere in C è a tutti gli effetti un tipo intero, questo vuol dire che può essere utilizzato in ogni espressione dove è lecito utilizzare un tipo intero. Ad esempio è possibile sommare caratteri come mostra la seguente porzione di programma:

```
char c = 'a';
printf("%c%c%c\n",   c, c+1, c+2);  /* stampa abc      */
printf("%d %d %d\n", c, c+1, c+2);  /* stampa 97 98 99 */
```

La prima istruzione assegna come valore alla variabile `c` di tipo `char` il carattere `'a'` mentre le altre due stampano il valore delle espressioni "`c`", "`c+1`" e "`c+2`". La stampa è di nuovo effettuata con l'ausilio della funzione `printf()` ma in una forma più complessa di quelle viste fino ad ora. La funzione prende adesso quattro argomenti: il primo è una stringa di caratteri che specifica cosa stampare, gli altri tre sono le espressioni di cui si vuole stampare il valore. Il valore delle espressioni è stampato utilizzando le informazioni fornite nella stringa, in questo caso solo le diretive su come interpretare i valori delle espressioni. Avremo modo di parlare più diffusamente del Input/Output, per il momento ci basta sapere che ciascuna coppia di caratteri "`%`" seguito da un altro carattere indica una direttiva di conversione ossia specifica come deve essere interpretata la sequenza di bits che rappresenta il valore dell'espressione. In questo caso "`%c`" indica che questa deve essere interpretata come un carattere mentre "`%d`" indica che la sequenza rappresenta un intero. Le direttive vengono applicate in sequenza: la prima direttiva alla prima espressione `c`, la seconda alla seconda espressione `c+1` e la terza alla terza espressione `c+2`. Il risultato è quindi che la prima istruzione di stampa stampa il valore delle espressioni come caratteri mentre la seconda ne stampa il valore numerico intero secondo il codice ASCII.

Nonostante queste similitudini vi sono tuttavia alcune differenze tra i tipi interi e quelli carattere. Ad esempio il tipo `char` senza il qualificatore `signed` o `unsigned`, chiamato anche *plain char*, può essere considerato dal compilatore come:

- Equivalente a `signed char`
- Equivalente a `unsigned char`
- Tipo *pseudo-unsigned* (non Standard C), ossia può assumere solo valori non-negativi ma viene trattato come se fosse un tipo `signed` in tutte le operazioni e conversioni.

Un'altra arbitrarità del tipo carattere è la sua dimensione. Usualmente tutti i tipi carattere usano 1 byte, ovvero 8 bits, di memoria per cui un `signed char` può rappresentare i numeri interi da -128 a 127 su sistemi con rappresentazione in complemento a due, mentre un `unsigned char` i numeri interi da 0 a 255. Tuttavia in alcuni (pochi) casi vengono usati 7 bits o 9 bits o anche di più. Lo standard ISO C richiede che le specifiche dei tipi carattere siano definite nel file di header `limits.h` attraverso delle macros:

Nome	**Descrizione**	**Esempio**
`CHAR_BIT`	numero di bit in `char`	8
`SCHAR_MIN`	valore minimo `signed char`	-127
`SCHAR_MAX`	valore massimo `signed char`	127
`UCHAR_MAX`	valore massimo `unsigned char`	255

2.4.3. Tipo Booleano (C99)

Il C89 non ha un tipo specifico per rappresentare i valori Booleni "vero" (*true*) e "falso" (*false*) ma utilizza a questo scopo un qualsiasi tipo intero associando al valore Booleano "`false`" il valore intero 0 ed al valore Booleano "`true`" un qualsiasi valore intero non nullo, sia positivo

che negativo.

Il C99 ha introdotto il tipo intero senza segno `_Bool` che può assumere solo i valori "0" (`false`) e "1" (`true`). Per comodità il file di header `stdbool.h` definisce le macros

Macro	Valore
`bool`	`_Bool`
`false`	`0`
`true`	`1`

che possono essere utilizzate per evidenziare il carattere booleano del tipo. Il nome `bool` non è una keyword per evitare possibili conflitti in programmi pre-C99 che utilizzino `bool` come identificatore.

2.4.4. Tipo Floating-Point

Per rappresentare numeri reali come ad esempio 1.2, 3.14159 o −3.7 il C usa i tipi floating-point (virgola mobile) per questo i numeri reali sono anche chiamati numeri floating-point. Il nome floating-point viene dal tipo di codifica usata internamente per la rappresentazione dei numeri reali. Per distinguere tra i numeri interi e reali il C usa il punto decimale ".", così ad esempio "1" è un numero intero mentre "1.0" o "1." sono numeri reali.

Come per i numeri interi anche per i numeri reali il C fornisce tre tipi di differente lunghezza:

Tipo floating-point:

```
float
double
long double
```

I tipi floating-point sono sempre con segno. Il C non fissa le dimensioni dei tipi floating-point ne richiede che queste siano effettivamente differenti, il numero di bits assegnata ad ogni tipo dipende quindi dal sistema. Su molti computers il tipo `float` (singola precisione) usa 4 bytes mentre il tipo `double` (doppia precisione) usa 8 bytes. Il tipo `long double` su alcuni sistemi ha la stessa dimensione del tipo double, su altri ha una dimensione più grande e fornisce una precisione più alta. In ogni caso si può assumere che i numeri rappresentabili con il tipo `float` siano un sottoisieme di quelli rappresentabili con tipo `double` che a loro volta sono un sottoinsieme di quelli rappresentabili con il tipo `long double`. Lo Standard C richiede che le specifiche dei numeri floating-point siano definite nel file di header `float.h`.

Nel Traditional C tutti i numeri `float` erano trasformati automaticamente in `double` prima di essere utilizzati in qualsiasi operazione, con una conseguente possibile perdita di velocità di esecuzione del programma. Lo Standard ISO C non richiede più questa conversione ma lascia la libertà di effettuarla o no. Su alcuni compilatori è possibile attivare o disattivare la conversione mediante *flags* opportuni.

Rappresentazione floating-point

Nella rappresentazione floating-point ogni numero x viene scritto come:

$$x = b^e \times M$$

dove

b	base o *radix* della rappresentazione, di solito 2, 8 o 16
e	esponente della rappresentazione
M	mantissa

I valori dell'esponente e della mantissa sono codificati utilizzando due sottosequenze della sequenza di bits usati per il tipo floating-point, ad esempio con 16 bits si potrebbero usare 6 bits per l'esponente e 10 bits per la mantissa. Se adesso quindi volessimo codificare i numeri interi positivi in rappresentazione floating-point basterebbe usare per la mantissa e l'esponente una codifica `unsigned int` con rispettivamente 6 bits e 10 bits, per cui con base $b = 2$ si avrebbe ad esempio

bits	e	M	valore
000000 | 0000000000	000000	0000000000	$2^0 \times 0 = 0_{10}$
000000 | 1000000101	000000	1000000101	$2^0 \times 517 = 517_{10}$
001000 | 0100101101	001000	0100101101	$2^8 \times 301 = 77056_{10}$
111111 | 1111111111	111111	1111111111	$2^{63} \times 1023 = 9423991637655520332924_{10}$

Osserviamo che il numero positivo massimo rappresentabile in questo modo è molto più grande del numero intero massimo rappresentabile con codifica `unsigned int` a 16 bits: $2^{16} - 1 = 65535_{10}$. Tuttavia usando solo 10 bits per la mantissa la precisione passa da 2^{-17} a 2^{-11}, poiché numeri che differiscono sull'undicesimo bit sono considerati uguali. Inoltre il numero di valori differenti che possono essere rappresentati è minore. Ad esempio ci sono 64 modi diversi per rappresentare lo 0.

Rappresentazione fixed-point

Per rappresentare i numeri con il punto decimale di solito si usa per la mantissa una codifica fixed-point (virgola fissa) in cui alcuni dei bits più a destra (*less significative bits*) sono interpretati come potenze negative della base b. Il nome fixed-point viene dal fatto che il numero di bits interpretati come potenze negative è fisso e quindi anche il numero di digits dopo il punto decimale. Il numero di bits usati per le potenze negative, e quindi la posizione del punto decimale, dipende dal computer. Se, ad esempio, con una mantissa di 16 bits si usano i 15 bits meno significativi per le potenze negative con $b = 2$ si ha:

bits	valore
0 | 000000000000000	$0 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-1} + \cdots + 0 \times 2^{-15} = 0.000000000000000_{10}$
0 | 100000000000000	$0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-1} + \cdots + 0 \times 2^{-15} = 0.500000000000000_{10}$

$1\,|\,100000000000000 \quad 1\times 2^0+1\times 2^{-1}+0\times 2^{-1}+\cdots+0\times 2^{-15}=1.500000000000000_{10}$
$1\,|\,010000000000000 \quad 1\times 2^0+0\times 2^{-1}+1\times 2^{-1}+\cdots+0\times 2^{-15}=1.250000000000000_{10}$
$1\,|\,111111111111111 \quad 1\times 2^0+1\times 2^{-1}+1\times 2^{-1}+\cdots+1\times 2^{-15}=1.999969482421875_{10}$

I bits utilizzati per le potenze positive e negative sono stati separati per semplificare la lettura.

Esistono molte codifiche floating-point per i numeri reali che differiscono per la scelta della lunghezza della mantissa e dell'esponente, della posizione del punto decimale nella mantissa, della rappresentazione dei numeri negativi, l'uso di bits nascosti, etc. La codifica utilizzata dipende in generale dal computer, tuttavia su molti computers un numero floating-point x può essere scritto come:

$$x = s \times b^e \times \sum_{k=1}^{p} d_k \times b^{-k}, \qquad e_{\min} \le e \le e_{\max}$$

dove

- s segno (± 1) del numero
- b base o *radix* della rappresentazione, di solito 2, 8 o 16
- e esponente della rappresentazione
- p numero di b-digits a potenze negative
- d_k valore dei b-digits a potenze negative, $0 \le d_k < b$

dipendono in generale dal sistema. Su molti sistemi che usano 32 bits per il tipo `float` e 64 bits per il tipo double si ha rispettivamente $p = 24$, $e_{\min} = -125$ ed $e_{\max} = 128$ per il tipo `float` e $p = 53$, $e_{\min} = -1021$ ed $e_{\max} = 1024$ per il tipo `double`. Questo, con $b = 2$, permette di rappresentare con il tipo `float` (singola precisione) numeri reali nell'intervallo tra 10^{-38} e 10^{38} con una precisione di circa 6 cifre decimali, il che vuol dire che il numero reale x è rappresentato (a meno del segno e almeno in forma approssimata) come

$$x = 0.d_1d_2d_3d_4d_5d_6 \times 10^e, \qquad d_1 \neq 0$$

dove ogni d_i è un digit decimale, con d_1 non nullo, ed e è un intero compreso nell'intervallo tra -38 e $+38$. Analogamente il tipo `double` (doppia precisione) ha una precisione di circa 15 cifre decimali con l'intero e nell'intervallo tra -308 e $+308$.

Lo Standard C richiede che le specifiche dei tipi floating-point siano definite nel file di header `float.h` attraverso delle macros:

Nome	**Descrizione**	**Esempio**
`FLT_RADIX`	base rappresentazione b	2
`FLT_MANT_DIG`	p tipo `float`	24
`FLT_DIG`	cifre significative tipo `float`	6
`FLT_MIN_EXP`	$e_{\min}$ tipo `float`	-125
`FLT_MIN_10_EXP`	$n_{\min}$ tipo `float`	-37
`FLT_MAX_EXP`	$e_{\max}$ tipo `float`	128
`FLT_MAX_10_EXP`	$n_{\max}$ tipo `float`	38

Nome	Descrizione	Esempio
`DBL_MANT_DIG`	p tipo `double`	53
`DBL_DIG`	cifre significative tipo `double`	15
`DBL_MIN_EXP`	$e_{\min}$ tipo `double`	-1021
`DBL_MIN_10_EXP`	$n_{\min}$ tipo `double`	-307
`DBL_MAX_EXP`	$e_{\max}$ tipo `double`	1024
`DBL_MAX_10_EXP`	$n_{\max}$ tipo `double`	308
`LDBL_MANT_DIG`	p tipo `long double`	113
`LDBL_DIG`	cifre significative tipo `long double`	33
`LDBL_MIN_EXP`	$e_{\min}$ tipo `long double`	-16381
`LDBL_MIN_10_EXP`	$n_{\min}$ tipo `long double`	-4931
`LDBL_MAX_EXP`	$e_{\max}$ tipo `long double`	16384
`LDBL_MAX_10_EXP`	$n_{\max}$ tipo `lonng double`	4932

2.4.5. Tipo Complesso (C99)

Per poter manipolare i numeri complessi il C99 introduce il tipo complesso

Tipo Complesso:

```
float _Complex
double _Complex
long double _Complex
```

Questo tipo è un tipo floating-point, rappresentato come un array (vettore) di due elementi di tipo floating-point corrispondente, le cui specifiche sono definite nel file di header `complex.h`. Il tipo è specificato con la keyword `_Complex` e non `complex` per limitare possibili conflitti con l'utilizzo dell'identificatore `complex` in programmi pre-C99.

Il C99 prevede anche il tipo opzionale puramente immaginario

Tipo Immaginario:

```
float _Imaginary
double _Imaginary
long double _Imaginary
```

rappresentato da un solo elemento di tipo floating-point del tipo corrispondente. La presenza del tipo `_Imaginary` non è richiesta obbligatoriamente dal C99 ed è lasciata come opzionale.

2.5. Unità di Memoria (Rev. 2.1.5)

In C tutti i dati[1] sono rappresentati in memoria utilizzando un numero intero di *unità di memoria*. Ogni unità di memoria è costituita a sua volta da un numero fissato di *bits* ciascuno

[1] I campi di tipo bit fields di una struttura od unione sono un'eccezione a questa regola in quanto sono definiti specificando esplicitamente la loro dimensione in termini di numero di bit.

dei quali può assumere solo due valori: il valore `0` o `1`. Il numero di bits di in una unità di memoria dipende dal sistema ma in ogni caso deve essere tale che ogni unità di memoria sia univocamente identificata da un *indirizzo di memoria* e sia sufficientemente grande da contenere la rappresentazione di ognuno dei caratteri di base. Di conseguenza una unità di memoria è la quantità di memoria occupata da un oggetto di tipo `char`. Lo Standard C richiede che il numero di bits utilizzati per rappresentare un oggetto di tipo `char` sia specificato dalla macro `CHAR_BIT` definita nel file di header `limits.h`.

Osserviamo che spesso nonostante che il termine *byte* indichi una unità di 8 bits le unità di memoria in C sono chiamate genericamente *bytes* indipendentemente dal numero di bits che le costituiscono.

Per definizione la *dimensione* di un oggetto è il numero di bits utilizzati per rappresentarlo, ed è quindi uguale al numero di bits di memoria occupati dall'oggetto. Tutti gli oggetti di un dato tipo occupano la stessa quantità di memoria per cui possiamo definire la dimensione di un tipo come la quantità di memoria occupata da un qualsiasi oggetto di quel tipo. In C la memoria viene assegnata (*allocata*) in multipli di unità di memoria, per cui la dimensione degli oggetti viene misurata in di unità di memoria occupate. Di conseguenza la dimensione del tipo `char`, ovvero di un qualsiasi oggetto di tipo `char`, è per definizione uguale a `1`. Un gruppo di una o più unità di memoria contigue viene chiamato usualmente una *locazione di memoria*.

Per conoscere la dimensione di un tipo o di un oggetto il linguaggio C fornisce l'operatore

```
sizeof(T)
```

il cui valore è uguale al numero di bytes, ossia il numero di unità di memoria, utilizzati per rappresentare un oggetto qualsiasi di tipo `T`. Il tipo del valore restituito dall'operatore `sizeof` dipende dal sistema, in genere questo è `unsigned int` o `unsigned long int`.

Il seguente semplice programma mostra l'uso dell'operatore `sizeof` per conoscere le dimensioni dei principali tipi di base.

Programma: `type_size.c`

```c
/*********************************************/
/* Stampa la dimensione dei tipi fondamentali */
/*********************************************/
#include <stdio.h>

int main(void)
{
  printf(" The size of the following types is:\n");
  printf("       char: %3u bytes\n", sizeof(char));
  printf("      short: %3u bytes\n", sizeof(short));
  printf("        int: %3u bytes\n", sizeof(int));
  printf("       long: %3u bytes\n", sizeof(long));
  printf("   unsigned: %3u bytes\n", sizeof(unsigned));
  printf("      float: %3u bytes\n", sizeof(float));
  printf("     double: %3u bytes\n", sizeof(double));
  printf("long double: %3u bytes\n", sizeof(long double));
```

```
    return 0;
}
```

Il programma assume che il valore dell'operatore `sizeof` sia di tipo `unsigned int` ed inoltre lo stampa con al massimo 3 digits decimali per questo nella funzione `printf()` viene usata la *direttiva di conversione* "`%3u`". La direttive di conversione saranno trattate più diffusamente più avanti. Se il valore restituito dall'operatore `sizeof` non è di tipo `unsigned int` il compilatore può produrre un messaggio di *warning* per segnalare l'inconsistenza.

Il risultato di questo programma dipende chiaramente dal computer e dal compilatore, sul mio fornisce:

```
The size of the following types is:
      char:   1 bytes
     short:   2 bytes
       int:   4 bytes
      long:   4 bytes
  unsigned:   4 bytes
     float:   4 bytes
    double:   8 bytes
long double:  12 bytes
```

Nel seguito diremo che l'oggetto `a` è più piccolo o più corto dell'oggetto `b` se la dimensione del tipo di `a` è più piccola di quella del tipo di `b`. Analogamente diremo che l'oggetto `a` è più grande o più lungo dell'oggetto `b` se la dimensione del suo tipo è più grande di quella del tipo di `b`. Ad esempio nel precedente esempio un oggetto di tipo `int` è più lungo di uno di tipo `short` ma uguale ad uno di tipo `long`.

2.6. Costanti (Rev. 2.1.1)

Le costanti sono anche chiamate "*literals*" riservando il termine "costanti" per indicare oggetti il cui valore non può essere cambiato, oggetti costanti per l'appunto. Non faremo questa distinzione ed useremo il termine "costante" per entrambi.

Una costante è caratterizzata non solo dal suo valore ma anche dal tipo. Il linguaggio C ammette quattro tipi di costanti: intere, floating-point, carattere e stringa.

2.6.1. Costanti Intere

Le costanti intere sono sequenze di digits che rappresentano valori interi espressi in notazione decimale o ottale o esadecimale. Il tipo di notazione utilizzata viene specificata dai primi caratteri della sequenza secondo la convenzione:

- Se la costante intera inizia con il digit `0` (zero) non seguito dal carattere `x` o `X` viene interpretata come valore intero in notazione ottale (base 8) per cui i digits che seguono devono essere digits ottali

  0 1 2 3 4 5 6 7

- Se la costante intera inizia con il digit `0` (zero) seguito dal carattere `x` o `X` ("`0x`" o "`0X`") viene interpretata come un valore intero in notazione esadecimale (base 16) e quindi i digits che seguono devono essere digits esadecimali

  `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

  con in caratteri da `a` a `f`, o da `A` a `F`, che rappresentano i valori decimali da 10 a 15.

- Se la costante intera inizia con un digit decimale non nullo viene interpretata come un valore intero in notazione decimale ed i digits seguenti devono essere digits decimali

  `0 1 2 3 4 5 6 7 8 9`

Il valore di una costante intera, a meno che non sia più grande del massimo valore rappresentabile (*overflow*), è sempre non negativo per cui se la costante è preceduta dal segno meno "`-`" quest'ultimo è interpretato come un operatore applicato alla costante e non come parte della costante stessa.

Le costanti intere possono essere seguite dalla lettera `l` o `L` per indicare che la costante è di tipo `long`. È inoltre possibile aggiungere il suffisso `u` o `U` per indicare che la costante è senza segno. Il C99 introduce anche il suffisso `ll` o `LL` per indicare che la costante intera è di tipo `long long`. Siccome il carattere `l` può essere facilmente confuso con il digit `1` il suo uso è sconsigliato.

Qui di seguito riportiamo alcuni esempi di costanti intere

Costanti Intere:

```
17          /* costante intera decimale                    */
017         /* costante intera ottale                      */
0x17        /* costante intera esadecimale                 */
0           /* costante intera decimale                    */
00          /* costante intera ottale                      */
0x0         /* costante intera esadecimale                 */
-671239     /* costante intera decimale 'negativa'         */
138670L     /* costante intera decimale long               */
0178318U    /* costante intera ottale senza segno          */
0x178318UL  /* costante intera esadecimale senza segno long */
```

Il tipo effettivamente utilizzato per le costanti intere dipende dalla dimensioni, dalla notazione e dalla presenza o meno di suffissi secondo il principio di utilizzare il primo tipo intero di dimensione sufficiente per rappresentare la costante intera senza perdita di informazioni. Questo semplice principio si traduce però in una regola di attuazione differente tra il C89 ed il C99. Nello Standard C89 il tipo viene determinato secondo il seguente schema che riporta nella colonna a destra l'ordine con cui sono scelti i tipi interi per ciascuna delle forme delle costanti intere date nella colonna di sinistra:

Costante	Tipo Utilizzato

dd...d	`int`
	`long int`
	`unsigned long int`

`0`*dd...d*
`0X`*dd...d*
	`int`
	`unsigned int`
	`long int`
	`unsigned long int`

dd...d`U`
`0`*dd...d*`U`
`0X`*dd...d*`U`
	`unsigned int`
	`unsigned long int`

dd...d`L`
`0`*dd...d*`L`
`0X`*dd...d*`L`
	`long int`
	`unsigned long int`

dd...d`UL`
`0`*dd...d*`UL`
`0X`*dd...d*`UL`
	`unsigned long int`

Se il valore di una costante intera è più grande del massimo valore rappresentabile con il tipo `unsigned long int` si ha un *overflow* ed il risultato è indefinito.

2.6.2. Costanti Floating-Point

Le costanti floating-point rappresentano valori reali e nello Standard C89 sono sempre in notazione decimale. Queste sono scritte con il punto decimale o con un esponente con segno preceduto dal suffisso `e` o `E`, o con entrambi. Ad esempio le costanti floating-point `1.2e34` e `1.2e+34` entrambi indicano il valore 1.2×10^{34} mentre la costante floating-point `1.2e-34` ha come valore 1.2×10^{-34}. È possibile specificare il tipo utilizzato nella rappresentazione della costante floating-point utilizzando il suffisso `f` o `F` per indicare il tipo `float` ed il suffisso `l` o `L` per il tipo `long double`. Anche in questo caso si sconsiglia l'uso del suffisso `l` in favore del suffisso `L` per la sua somiglianza con il digit `1`. In assenza di suffisso le costanti floating-point sono sempre di tipo `double`.

Come per le costanti intere anche il valore delle costanti floating-point è sempre non negativo per cui un eventuale segno negativo "`-`" è considerato un operatore e non parte della costante.

Le seguenti sono alcuni esempi di costanti floating-point:

Costanti Floating-Point:

`1.0`	`1936.27`	`3.14156`
`1.E-02`	`0.4e-2`	`8e+3`
`5.8e50L`	`2.0f`	`-5.7F`

Se il valore di una costante floating-point è troppo piccolo o troppo grande per essere rappresentato il risultato è arbitrario. Alcuni compilatori segnalano il problema con un messaggio di

"warning", tuttavia la maggior parte semplicemente attribuisce valori arbitrati alla costante.

Il C99 permette di utilizzare anche la notazione esadecimale per le costanti floating-point precedendo l'esponente con il carattere `p` per evitare confusione con il digit esadecimale `e`. In questo caso però l'esponente è la rappresentazione binaria di un numero intero con segno che rappresenta una potenza di `2` e non `10` come nel caso della notazione decimale.

Sempre nel C99 è possibile utilizzare costanti complesse utilizzando la costante immaginaria `_Complex_I`. Per semplificare la scrittura nel file di header `complex.h` viene definita la macro `I` in sostituzione della costante complessa, per cui una costante complessa può essere scritta indipendentemente come `1.0 + 1.0*_Complex_I` o come `1.0 + 1.0*I`, in entrambi i casi il valore è il valore complesso `1+i`. Se si usa la seconda scrittura bisogna però ricordarsi di includere nel file il file di header `complex.h`.

2.6.3. Costanti Carattere

Le costanti carattere rappresentano un singolo carattere e sono sempre scritte racchiudendo il carattere tra due apostrofi "'", come ad esempio

```
'a' 'b' '+' 'G'
```

Qualsiasi carattere può essere utilizzato in un costante carattere, tuttavia non tutti i caratteri possono essere inseriti direttamente nella costante come ad esempio il *return* o il *newline*. In questi casi i caratteri devono essere inseriti usando i caratteri di *escape*.

I caratteri di escape si dividono in due classi: i caratteri di escape "carattere" ed i caratteri di escape "numerici". I primi permettono di rappresentare alcuni caratteri di formattazione e caratteri speciali mentre i secondi permettono di specificare i caratteri con il codice numerico della rappresentazione. Tutti i caratteri di escape iniziano con il carattere "\" (*backslash*).

Caratteri di escape carattere

I caratteri di escape carattere permettono di rappresentare alcuni caratteri speciali e di formattazione in una forma indipendente dal tipo di codifica utilizzata dal sistema per i caratteri. I caratteri di escape carattere sono formati dal carattere "\" seguito da uno dei seguenti caratteri:

Carattere	Carattere Escape	Significato
a	\a	alert
b	\b	backspace
f	\f	form feed
n	\n	newline
r	\r	return
t	\t	horizontal tab
v	\v	vertical tab
\	\\	backslash
'	\'	apostrofo

"	\"	doppi apici
?	\?	p. interrogativo

Come visto precedentemente il carattere di escape "\?" viene utilizzato quando vi possono essere problemi di interpretazione con i trigraph. Se il carattere "" compare in una costante carattere il backslash può essere omesso.

Caratteri di escape numerici

I caratteri di escape numerici permettono di specificare i caratteri direttamente codice il loro codice numerico. Differentemente dai codici di escape carattere i codici di escape numerici dipendono quindi dal codice utilizzato. I caratteri di escape numerici sono formati dal carattere "\" seguito dal codice numerico del carattere nella codifica utilizzata. Lo Standard C permette di utilizzare sia il codice in formato ottale che quello in formato esadecimale. Ad esempio nel codice ASCII il carattere `'a'` può essere scritto sia come `'\141'` (codice ottale) che come `'\x61'` (codice esadecimale). Altri esempi sempre con il codice ASCII sono

`'\0'`		⇒	carattere nullo
`'\12'`	`'\x0a'`	⇒	*newline*
`'\134'`	`'\x5c'`	⇒	*backslash*
`'\62'`	`'\x32'`	⇒	carattere 2

Il carattere nullo viene sempre scritto come '\0'. Se il codice numerico non corrisponde a nessun carattere il risultato dipende dal sistema.

Quando si utilizzano caratteri di escape numerici è bene ricordare che i codici numerici dipendono dalla codifica dei caratteri utilizzata con conseguenze evidenti sulla portabilità del programma.

Se il carattere che segue il backslash non è un digit ottale o il carattere `x` o uno dei caratteri utilizzati per i caratteri di escape carattere il risultato è indefinito.

Le costanti carattere sono sempre di tipo `int` ed il loro valore è uguale al valore intero decimale associato al carattere nella codifica utilizzata. Ad esempio nel codice ASCII la costante carattere `'a'` ha valore 97 mentre la costante carattere `'\\'` che rappresenta il backslash ha il valore 92. Il valore non dipende da come viene scritto il carattere per cui il valore della costanti carattere `'a'`, `'\141'` e `'\x61'` è sempre 97.

Alcuni compilatori permettono costanti carattere con più di un singolo carattere, ad esempio `'ab'`. Sebbene questo sia permesso dallo Standard C l'uso è sconsigliato poiché può dare problemi di compatibilità. In questo caso è consigliabile utilizzare delle costanti stringa.

Infine osserviamo che per permettere di rappresentare caratteri non rappresentabili dal tipo `char`, ad esempio quelli dell'alfabeto Giapponese, lo Standard C fornisce delle costanti caratteri di dimensione più grande chiamate *wide characters* ed ottenute precedendo la costante carattere con la lettera `L`. Una costante wide characters tipicamente è composta da una sequenza di caratteri e caratteri di escape che insieme formano un singolo carattere *multibyte*. In genere la corrispondenza tra i caratteri multibyte ed i caratteri wide dipende dal sistema. In

questo primer non considereremo questo tipo di caratteri ma assumeremo sempre di utilizzare carattere codificati secondo il codice ASCII.

2.6.4. Costanti Stringa

Una costante stringa è una sequenza, anche vuota, di caratteri racchiusa tra due doppi apici "". Come per le costanti carattere i caratteri non direttamente inseribili sono ottenuti con i caratteri di escape. Il carattere apostrofo "'" può essere utilizzato nelle costanti stringa senza il backslash. Lo Standard C permette di utilizzare caratteri wide nelle stringhe, in questo caso la stringa deve essere preceduta dalla lettera `L` (*wide strings*).

Di seguito riportiamo alcuni esempi di stringhe

Esempi:

```
"una␣costante␣stringa"
"a␣=␣b␣+␣c;"                          /* stringa non operazione  */
"␣␣␣␣␣"                                /* una stringa di spazi    */
""                                     /* la stringa nulla        */
"questa␣contiene␣\"␣il␣doppio␣apice"   /* carattere speciale      */
"questa␣contiene␣\␣il␣backslash"       /* carattere speciale      */
"questa␣contiene␣\n␣il␣newline"        /* carattere speciale      */
"/*␣stringa␣non␣commento␣*/"           /* stringa non commento    */
"a"                                    /* stringa di un carattere */
```

Le costanti stringa sono trattate in modo diverso dalle costanti carattere poiché per ogni costante stringa di n caratteri viene riservato uno spazio per contenere $n + 1$ caratteri: gli n caratteri della stringa più il carattere nullo '\0' posto alla fine della stringa per indicarne la fine. Di conseguenza la costante stringa di un carattere `"a"` e la costante carattere `'a'` sono *diverse*.

Nello Standard C due costanti stringa adiacenti separate solo da spazi bianchi o *newline* sono automaticamente concatenate in un'unica costante stringa più lunga. Ad esempio le due costanti stringa

```
"abc" "ABC"
```

vengono concatenate nell'unica costante stringa

```
"abcABC"
```

Il carattere '\0' di fine stringa alla fine della prima costante stringa viene eliminato. Questa proprietà permette di scrivere facilmente costanti stringa piuttosto lunghe dividendole su più linee. Una costante stringa normale ed una contenete caratteri wide non possono essere concatenate in questo modo.

2.7. Dichiarazione di variabili (Rev. 2.1.3)

Nel linguaggio C, diversamente da altri linguaggi, tutti gli oggetti devono essere dichiarati. Dichiarare un oggetto in C significa specificarne le sue caratteristiche ed associargli un identi-

ficatore o nome. È una buona norma di programmazione usare identificatori che rispecchino il più possibile il significato o le caratteristiche dell'oggetto, come pure aggiungere brevi commenti descrittivi all'atto della dichiarazione di un oggetto. Le dichiarazioni, oltre che ad identificare i vari oggetti con dei nomi, servono ad informare il compilatore C su come l'oggetto debba essere trattato; ad esempio quali valori può prendere, l'insieme di operazioni che possono essere effettuate, quanta memoria debba essergli riservata e così via. Normalmente il linguaggio C richiede che tutti gli identificatori siano completamente specificati prima del loro utilizzo, vi sono tuttavia casi in cui un identificatore può essere utilizzato prima che la dichiarazione sia stata completata. In questi casi si parla di *forward declaration* o *forward reference*. Incontreremo situazioni di questo genere più avanti quando tratteremo oggetti complessi come funzioni o strutture.

Una descrizione generale delle dichiarazioni in C è piuttosto difficile perché a seconda delle caratteristiche degli oggetti considerati le dichiarazioni possono essere piuttosto involute e complicate anche dal fatto che la sintassi non è sempre delle più trasparenti. Fortunatamente nel caso delle variabili le dichiarazioni prendono forme relativamente semplici.

Una variabile è un oggetto in cui è possibile memorizzare informazioni. Il termine "variabile" deriva dal fatto che in generale le informazioni memorizzate possono essere cambiate. La forma più semplice di una dichiarazione di variabile è

```
type var_name;    /* commento opzionale */
```

dove `type` specifica il tipo `T` delle informazioni contenute nella variabile e l'identificatore `var_name` è il nome che identifica la variabile. Il punto e virgola "`;`" indica in questo caso la fine della dichiarazione. Esempi di dichiarazioni di variabili sono

```
int totale;
float media;
```

La prima dichiarazione dichiara che `totale` è l'identificatore di una variabile che contiene dati di tipo intero `int`. Questo vuol dire che qualsiasi cosa si trovi memorizzato nella variabile di nome `totale` viene interpretato come tipo `int` e trattato di conseguenza. Siccome l'identificatore identifica univocamente la variabile possiamo semplicemente dire che "`totale` è una variabile di tipo `int`". Analogamente possiamo dire che la seconda dichiarazione dichiara la variabile `media` di tipo `float`, ovvero che `media` è l'identificatore di una variabile che contiene dati di tipo floating-point `float`.

Chiaramente non è possibile utilizzare lo stesso identificatore per due variabili differenti, anche se con opportune restrizioni è possibile utilizzare lo stesso identificatore per oggetti diversi. In questo caso si parla di *overloading* dell'indentificatore.

Lo Standard C89 permette di omettere lo specificatore di tipo `type` nel qual caso viene assunto per *default* il tipo `int`. Omettere lo specificatore di tipo è tuttavia considerato un pessimo stile di programmazione ed infatti nello Standard C99 viene considerato un errore.

Più variabili dello stesso tipo possono essere dichiarate con una stessa istruzione separando gli identificatori con la virgola "`,`" come ad esempio nella dichiarazione seguente

```
long int gran_totale, parziale;
```

che dichiara le due variabili `gran_totale` e `parziale` di tipo `long int`.

2.7.1. Inizializzazione

La dichiarazione di una variabile riserva uno spazio di memoria di dimensione appropriata a contenere il tipo della variabile ma non necessariamente gli assegna un valore, ossia modifica il contenuto della zona di memoria assegnata. Una variabile dichiarata ma a cui non sia stato assegnato un valore è detta *non inizializzata* ed il suo valore è in generale indefinito poiché dipende da quello che si trova in quel momento nella zona di memoria assegnatagli.[2] L'utilizzo di una variabile non inizializzata produce quindi un risultato indefinito.

Per assegnare un valore ad una variabile si deve utilizzare l'operatore di assegnamento "`=`" come mostrato dalle due seguenti istruzioni

```
unsigned short int i;
i = 9;
```

che dichiarano la variabile `i` di tipo `unsigned short int` e gli assegnano il valore `9`. Il valore assegnato alla variabile ricopre qualsiasi valore, definito o no, precedentemente contenuto in essa. Quando ad una variabile viene assegnato un valore per la prima volta si dice che la variabile viene inizializzata.

È possibile assegnare un valore ad una variabile contemporaneamente alla sua dichiarazione, ad esempio le due precedenti istruzioni sono equivalenti a

```
unsigned short int i = 9;
```

Sebbene la dichiarazione e l'inizializzazione della variabile compaiano in una sola istruzione, queste sono considerate come due operazioni separate: prima la dichiarazione della variabile e poi l'assegnazione del valore.

2.7.2. Qualificatore `const`

A volte può risultare utile che il valore di una variabile non possa essere cambiato. Questo si ottiene specificando il *qualificatore* "`const`" nella dichiarazione,

```
const type var_name = value;
```

dove `type` è il tipo `T` della variabile. Da momento che il valore della variabile non può essere modificato la variabile deve essere inizializzata contestualmente alla dichiarazione.

Ad esempio per dichiarare una variabile che contenga il valore di π ed il cui valore non possa essere modificato si può usare l'istruzione:

```
const double pi_greco = 3.1415927;
```

La presenza del qualificatore `const` implica che il valore della variabile `pi_greco` non può essere cambiato, per cui un'eventuale istruzione del tipo

```
pi_greco = 6.28;        /* Illegale */
```

[2] Come vedremo più avanti il comportamento dipende dalla classe di memorizzazione della variabile. Le variabili in classe di memorizzaione *automatica* non vengono inizializzate mentre quelle in classe *statica* sono automaticamente inizializzate a zero.

avrebbe come conseguenza un messaggio di errore in compilazione.

Le variabili il cui valore non può essere cambiato sono chiamate anche "costanti", da non confondersi però con le costanti nel senso di litterals discusse precedentemente.

Per ridurre le possibilità di errore spesso si usa una convenzione per i nomi delle costanti, ad esempio utilizzando identificatori con una o più caratteri maiuscoli.

2.7.3. Qualificatore `volatile`

A volte può accadere che il valore di una variabile possa essere modificato in un modo non controllato dal programma. Ad esempio il valore di una variabile che contiene lo stato del lettore CD viene modificato dall'hardware che gestisce il lettore. In questi casi si utilizza il qualificatore `volatile` per informare il compilatore C che l'oggetto può essere modificato in un modo non sotto il controllo del programma.

Una variabile volatile viene dichiarata aggiungendo il qualificatore `volatile` alla dichiarazione:

```
volatile type var_name;
```

dove `type` è il tipo `T` della variabile.

Il qualificatore `const` ed il qualificatore `volatile` possono essere specificati contemporaneamente, ad esempio la dichiarazione

```
const volatile int cd_ready;
```

dichiara la variabile `cd_ready` di tipo `volatile int` il cui valore non può essere modificato nel programma. Questo non crea problemi perché se il valore della variabile `cd_ready` indica lo stato del lettore CD questo viene modificato dall'hardware che gestisce il lettore. Il fatto poi che la variabile sia dichiarata anche di tipo `const` significa che il valore della variabile non può essere modificato all'interno del programma perché, ad esempio, il programma non gestisce il lettore e quindi non deve poter modificare il valore della variabile `cd_ready`.

Oggetti di tipo `volatile` sono utilizzati nella scrittura del *driver* dei devices per accedere a zone di memoria con informazioni che possono essere cambiate dal sistema o dall'hardware che gestisce i devices.

2.8. Operatori (Rev. 2.1.1)

I dati contenuti nelle variabili possono essere manipolati utilizzando operatori e funzioni. Una sequenza valida di variabili, costanti, operatori, separatori e funzioni viene chiamata *espressione*. Un'espressione terminata dal punto e virgola "`;`" forma un'*istruzione*.

2.8.1. Operatori matematici

Nel linguaggio C le operazioni matematiche di somma, sottrazione, moltiplicazione e divisione sono effettuate mediante gli operatori matematici:

Operatore	Operazione
`+`	somma
`-`	sottrazione
`*`	moltiplicazione
`/`	divisione
`%`	modulo

Tutti questi operatori, ad eccezione dell'operatore modulo che opera solo su tipi interi, operano sia su tipi interi che su tipi floating-point.

Osserviamo che il significato di alcuni dei caratteri utilizzati per gli operatori matematici possono dipendere dal contesto. Ad esempio nell'istruzione

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

il carattere "`%`" indica l'inizio di una direttiva di stampa, nell'istruzione seguente

```
a = b % 2;
```

il carattere "`%`" indica l'operatore modulo che fornisce il resto della divisione del valore della variabile `b` per `2`. Anche il significato del carattere "`*`" può dipendere dal contesto, come vedremo più avanti quando introdurremo i puntatori.

Gli operatori matematici sono *operatori binari* perché operano su due operandi. Il *valore* di un operatore matematico è il valore del risultato dell'operazione. Gli operandi possono essere semplici variabili, come in

```
int a = 1, b = 2;
int c;
c = a + b
```

oppure espressioni

```
int a = 1, b = 2;
int c;
c = a * (a + b);
```

In ogni caso il valore di ciascun operando è valutato prima di effettuare l'operazione, per cui nell'esempio precedente viene prima calcolato il valore di `a` e di `a+b` e poi i due valori vengono moltiplicati. Tuttavia l'ordine con cui vengono calcolati i valori dei due operandi, ossia se prima quella di sinistra o prima quello di destra, non è fissato.

Gli operatori "`-`" e "`+`" possono comparire anche in forma *unaria*, ossia prendere un solo operando, come nel seguente esempio

```
int a = 1, b;
b = -a;
```

Anche in questo caso l'operando può essere una semplice variabile, come nell'esempio, o una costante o un'espressione. L'espressione unaria "`-expr`" è interpretata come "`0 - (expr)`". Il risultato di questa espressione può essere indefinito se l'operando `expr` è di tipo intero

`signed` o floating-point e l'operazione produce un *overflow*. Nel caso in cui l'operando `expr` sia di tipo intero `unsigned` il risultato è ancora di tipo `unsigned`, e quindi non negativo, ed è uguale a $2^n - $ `expr` dove n è il numero di bits utilizzati per rappresentare il risultato. Sebbene questo possa sembrare strano osserviamo che la rappresentazione binaria con n bits di 2^n non differisce da quella di 0 per cui anche se il valore è positivo l'uguaglianza `expr` $+ (-$`expr`$) = 0$ è sempre soddisfatta. Osserviamo inoltre che se si utilizza la codifica in complemento a due la rappresentazione binaria `unsigned` di $2^n - $ `expr` coincide con la codifica in complemento a due di `-expr`.

L'operatore unario "`+`" è stato introdotto per simmetria. L'espressione unaria "`+expr`" è equivalente a "`0 + (expr)`".

2.8.2. Precedenza ed associatività

Per valutare le espressioni il compilatore C usa le regole di precedenza ed associatività degli operatori che specificano l'ordine con cui valutare le varie (sotto)espressioni che compongono l'espressione principale. La *precedenza* indica l'ordine in cui le operazioni specificate dai vari operatori vanno eseguite. Ad esempio l'operatore di moltiplicazione "`*`" ha un livello di precedenza più alto dell'operatore di somma "`+`" per cui le moltiplicazioni sono valutate prima delle somme. Di conseguenza l'espressione

```
1 + 2 * 3
```

ha come valore 7.

L'ordine con cui un'espressione viene valutata può essere alterato utilizzando le parentesi tonde "`()`". Queste hanno un livello di precedenza molto alto per cui le espressioni in parentesi vengono valutate per prime. Di conseguenza mentre

```
1 + (2 * 3)
```

ha come nel caso precedente valore 7, l'espressione

```
(1 + 2) * 3
```

ha valore 9. Nel primo caso infatti non si modifica l'ordine dato dalle regole di precedenza degli operatori per cui il risultato non cambia. Nel secondo caso invece la somma, con livello di precedenza più basso della moltiplicazione, viene eseguita prima a causa delle parentesi cosicché l'ordine viene modificato, ed anche il risultato.

L'*associatività* indica l'ordine con cui valutare le operazioni specificate da operatori con lo stesso livello di precedenza all'interno di una espressione. Ad esempio gli operatori matematici binari hanno associatività da *sinistra*, per cui quando in un'espressione compaiano operatori matematici con lo stesso livello di precedenza, l'espressione viene valutata associando gli operatori partendo da sinistra. Ad esempio la seguente espressione

```
1 + 2 + 3 - 4 + 5 + 6
```

viene valutata come

```
(((((1 + 2) + 3) - 4) + 5) + 6)
```

Gli operatori unari "+" e "-" hanno invece associatività da destra e priorità più alta rispetto agli operatori matematici binari.

2.8.3. Operatori di incremento "++" e decremento "--"

Il linguaggio C fornisce l'operatore unario di incremento "++" e l'operatore unario di decremento "--". Ciascun operatore forma ma un unico token per cui non vi possono essere spazi tra i due caratteri che lo compongono. Entrambi gli operatori prendono per operando variabili, ma non costanti o espressioni, e possono comparire sia in posizione *prefissa* (prima dell'operando) che *postfissa* (dopo l'operando).

```
i++                 /* operatore postfisso            */
--k                 /* operatore prefisso             */
8876--              /* Illegale  operando costante    */
++( a * 3 + b)      /* Illegale  operando espressione */
```

L'operatore di incremento "++", sia nella forma prefissa che in quella postfissa, incrementa di 1 il valore della variabile a cui è applicato, mentre l'operatore di decremento "--", sia prefisso che postfisso, ne diminuisce il valore di 1.

La differenza tra gli operatori in forma postfissa e quelli in forma prefissa è nel valore dell'operatore: nella forma prefissa il valore dell'operatore è quello della variabile su cui opera dopo l'incremento o decremento, mentre nella forma postfissa il valore è quello della variabile prima dell'incremento o decremento. Ad esempio sia `++i` che `i++` sono equivalenti a

```
i = i + 1
```

tuttavia il valore di `++i` è il valore di `i+1`, mentre quello di `i++` è il valore della variabile `i`, come mostrato dal seguente semplice programma:

```
int main(void)
{
  int a, b, c;

  c = 0;
  a = ++c;      /* c vale 1, a vale 1 = valore c dopo incremento  */
  b = c++;      /* c vale 2, b vale 1 = valore c prima incremento */

  printf("a: %d, b: %d, c: %d\n", a, b, ++c);
  return 0;
}
```

Quando il programma viene eseguito si ha

```
a: 1, b: 1, c: 3
```

Gli operatori nella forma postfissa e prefissa differiscono, oltre che per il valore, anche per l'associatività che è da sinistra per gli operatori nella forma postfissa e da destra per quelli nella forma prefissa.

Gli operatori "++" e "--" forniscono un modo molto compatto di indicare un incremento o decremento di una unità del valore di una variabile. Tuttavia se in situazioni semplici la

forma postfissa e prefissa è irrilevante, in altre bisogna fare molta attenzione a quale delle due è la corretta altrimenti i risultati potrebbero essere differenti da quelli voluti. Inoltre è buona norma evitare di scrivere le espressioni contenenti gli operatori di incremento/decremento in forme che possano risultare ambigue, come ad esempio

```
++ a * b + c --     /* equivale a: ((++a) * b ) + (c--) */

5 - - a / ++ b      /* equivale a: 5 - ( (-a) / (++b))  */
```

Altri problemi possono nascere a causa del fatto che gli operatori unari "`++`" e "`--`" modificano il valore dell'operando e possono quindi dare origine ad effetti collaterali (*side effects*). Si ha un *side effect* ogni qual volta in una espressione oltre all'operazione principale viene eseguita una operazione secondaria. Consideriamo ad esempio le seguente istruzione

```
y = x++;
```

Questa istruzione effettua due operazioni. La prima è l'operazione principale che assegna alla variabile `y` il valore della variabile `x`, la seconda che incrementa di uno il valore della variabile `x` è un'operazione secondaria ed è quindi un side effect.

La presenza di side effects può creare problemi specialmente in espressioni complesse. Consideriamo ad esempio le seguenti istruzioni

```
int value, var = 1;
value = (var++ * 2) + (var++ * 3);
```

Quale è il valore finale di `value`? La risposta è ambigua poiché dipende dall'ordine con cui i due operandi dell'operatore "`+`" vengono valutati. Supponiamo che venga eseguita prima l'espressione di sinistra che chiede di moltiplicare il valore di `var` per `2` e di aggiungere `1` al valore di `var`. Il risultato è quindi che il valore dell'operando di sinistra è `2`, mentre il valore di `var` è passato da `1` a `2`. L'espressione di destra chiede una cosa simile, ossia di moltiplicare per `3` il valore di `var` e di aggiungere `1` a `var`. Siccome adesso `var` vale `2` il valore dell'operando di destra è `6`, mentre il valore di `var` passa a `3`. Infine i valori dei due operandi vengono sommati tra di loro ed il risultato assegnato a `value`. Quindi se viene valutato prima l'operando a sinistra e poi quello a destra il valore di `value` è `2+6=8`. Se invece viene eseguita per prima l'espressione a destra dell'operatore "`+`" è facile convincersi che adesso il valore dell'operando di destra è `3` mentre quello dell'operando di sinistra è `4`, per cui il risultato è che il valore di `value` è `4+3=7`. Ma non è finita, alcuni compilatori applicano l'operatore di incremento (o decremento) solo dopo (o prima se in posizione prefissa) che tutte le espressioni siano state valutate. In questo caso il valore di `value` sarebbe `2+3=5`. Lo Standard C non richiede che le espressioni siano valutate in un ordine preciso, ma lascia libertà al compilatore di eseguirle nell'ordine che ritiene più efficiente. Inoltre l'ordine può variare a seconda del livello di ottimizzazione richiesto. La morale della storia è quindi quella di scrivere i programmi in modo semplice utilizzando un solo operatore "`++`" o "`--`" per istruzione evitando così costruzioni la cui valutazione può risultare ambigua.

2.8.4. Operatore di assegnamento semplice

Il modo più semplice per assegnare o modificare il valore di una variabile è con l'operatore binario di assegnamento "`=`" attraverso un'espressione del tipo

```
variable = expression
```

L'operando di sinistra è sempre una variabile mentre quello di destra può essere un'espressione qualsiasi il cui valore, dopo essere stato convertito se necessario nel tipo della variable, viene assegnato alla variabile. Abbiamo già incontrato più volte espressioni di questo tipo, ad esempio per inizializzare una variabile.

Diversamente da altri linguaggi in C "`=`" è un operatore, per cui non solo ha un livello di precedenza ed una regola di associatività ma anche un valore ed un tipo. Il livello di precedenza è molto basso mentre l'associatività è da destra. Questo vuol dire che l'espressione di assegnamento precedente viene valutata nell'ordine seguente: prima viene valutato il valore dell'espressione `expression` e poi assegnato con le dovute conversioni alla variabile `variable`. Il valore dell'operatore "`=`" è uguale al valore assegnato, ossia al valore dell'espressione `expression`, ed il tipo è uguale al tipo del suo operando di sinistra, ossia il tipo della variabile `variable`.

Per illustrare questo punto consideriamo le seguenti istruzioni

```
a = 2;
b = 3;
c = a + b;
```

che assegnano i valori `2`, `3` e `5` rispettivamente alle variabili `a`, `b` e `c`. Sfruttando il fatto che "`=`" è un operatore il cui valore è uguale al valore assegnato, lo stesso risultato si può ottiene con l'istruzione

```
c = (a = 2) + (b = 3);
```

Le parentesi sono necessarie altrimenti a causa dell'associatività da destra dell'operatore "`=`" l'espressione

```
c = a = 2 + b = 3;
```

verrebbe valutata come

```
c = (a = (2 + b = 3));
```

generando quindi un errore di compilazione del tipo "invalid lvalue in assignment" poiché a sinistra dell'ultimo operatore "`=`" si verrebbe a trovare un'espressione, "`2+b`", e non una variabile. Se vengono omesse le parentesi intorno all'espressione "`a = 2`", ma non quelle intorno a "`b = 3`", l'istruzione è sintatticamente corretta e non si avrebbe un errore di compilazione, ma nemmeno il risultato voluto. Infatti

```
c = a = 2 + (b = 3);
```

viene valutata come

```
c = (a = 2 + (b = 3));
```

con il risultato di assegnare il valore `3` alla variabile `b` ed il valore `5` alle variabili `a` e `c`. Questo è un errore di esecuzione, in generale ben più difficile da individuare dell'errore di compilazione.

Sfruttando le proprietà dell'operatore "`=`" è possibile effettuare assegnamenti multipli, come ad esempio

```
a = b = c = d = 0;
```

poiché questa istruzione viene valutata come

```
a = (b = (c = (d = 0)));
```

assegnando quindi il valore `0` a tutte e quattro le variabili.

È bene tenere in mente che le espressioni di assegnazione, sebbene abbiano una scrittura simile alle espressioni matematiche, sono distinte da esse. Ad esempio la scrittura

```
x + 1 = 0
```

è perfettamente valida come espressione matematica ma non come espressione di assegnamento. Al contrario la scrittura

```
x = x + 1
```

è una valida espressione di assegnamento ma non ha senso come espressione matematica.

2.8.5. Operatori di assegnamento composti

Oltre all'operatore di assegnamento semplice "`=`" il C ha anche operatori binari di assegnamento composti della forma "`op=`" dove "`op`" è uno dei cinque operatori matematici od uno dei cinque operatori bit-a-bit che saranno studiati più avanti:

Operatori assegnamento composti

```
+=   -=   *=   /=   %=   >>=   <<=   &=   ^=   |=
```

Nello Standard C gli operatori di assegnamento composti formano un unico token per cui non vi possono essere spazi bianchi tra l'operatore "`op`" ed il segno di uguale "`=`".

L'espressione di assegnamento composto

```
variable op= expression
```

può essere pensata come l'espressione di assegnamento

```
variable = variable op expression
```

con la restrizione che l'operatore "`op`" opera sul valore dell'espressione "`expression`" e non sull'espressione stessa. In altre parole in un'assegnazione con un operatore di assegnamento composto per prima cosa viene calcolato il valore dei due operandi di "`op=`" con i quali viene poi eseguita l'operazione indicata dall'operatore "`op`" ed infine il valore risultante viene assegnato all'operando di sinistra, ossia alla variabile `variable`.

Questa differenza non è importante per espressioni di assegnamento semplici come

```
i += 2          /* equivalente a i = i + 2 */
i -= 2          /* equivalente a i = i - 2 */
i /= 2          /* equivalente a i = i / 2 */
i *= 2          /* equivalente a i = i * 2 */
i %= 2          /* equivalente a i = i % 2 */
```

tuttavia in altri casi può essere all'origine di errori difficili da trovare. Ad esempio la seguente espressione

```
i *= j + 2
```

viene valutata come

```
i = i * (j + 2)
```

e non come

```
i = (i * j) + 2
```

ossia come sarebbe valutata l'espressione

```
i = i * j + 2
```

per cui in questo caso il risultato dell'espressione "`a op= b`" è differente dal risultato dell'espressione "`a = a op b`".

Gli operatori di assegnamento composto "`op=`" hanno lo stesso livello di precedenza ed associatività dell'operatore di assegnamento semplice "`=`". Inoltre, come per quest'ultimo, il valore è uguale al valore assegnato al suo operando di sinistra ed il tipo è quello del suo operando di sinistra.

2.8.6. Operatore "," ed espressioni sequenziali

Due espressioni separate dal carattere carattere '`,`' (virgola)

```
expression_1, expression_2
```

formano una *comma expression*. In questo contesto "`,`" è un operatore binario con associatività a sinistra e livello di precedenza più basso di qualsiasi altro operatore. L'operatore viene valutato valutando per prima l'espressione `expression_1` che figura come operando di sinistra dell'operatore "`,`" e poi l'espressione `expression_2` che figura come operando di destra. Il tipo e valore dell'operatore "`,`", ovvero della comma expression, è il tipo `T` ed il valore del suo operando di destra. Se l'operando di sinistra produce un valore questo viene semplicemente ignorato.

L'operatore "`,`" è associativo cosicché è possibile scrivere una singola espressione composta da più espressioni separate dall'operatore "`,`". Un'espressione di questo tipo viene chiamata espressione sequenziale perché le espressioni che la compongono sono valutate in ordine sequenziale a partire da quella più a sinistra. Il tipo ed il valore dell'espressione sequenziale è il tipo `T` ed il valore dell'ultima espressione più a destra. Ad esempio le istruzioni

```
expr_1;
expr_2;
expr_3;
var = exp_4;
```

possono essere scritte come una sola espressione sequenziale

```
var = (exp_1, exp_2, exp_3, exp_4);  /* Parentesi necessarie */
```

Le parentesi "`()`" sono necessarie perché l'operatore "`,`" ha un livello di precedenza inferiore all'operatore di assegnazione "`=`" per cui l'istruzione

```
var = exp_1, exp_2, exp_3, exp_4;
```

verrebbe interpretata come

```
(var = exp_1), exp_2, exp_3, exp_4;
```

Utilizzando l'operatore "`,`" è facile scrivere un'espressione il cui risultato è uguale a quello degli operatori "`++`" e "`--`" in forma postfissa. Ad esempio l'espressione `i++` è perfettamente equivalente a

```
(i += 1, i - 1)
```

ed analogamente `i--` può essere ottenuta come

```
(i -= 1, i + 1)
```

Queste "rappresentazioni" sono usate a volte al posto degli operatori di incremento "`++`"e decremento "`--`" in forma postfissa perché mostrano più chiaramente il risultato dell'espressione.

Come vedremo più avanti l'operatore "`,`" viene usualmente utilizzato nelle espressioni che controllano il cicli perché permette di "raggruppare" insieme più espressioni.

Nota: Il carattere '`,`' viene usato anche come separatore, ad esempio come per separare gli identificatori di due variabili dello stesso tipo nella dichiarazione

```
int i, j;
```

In questi casi il carattere '`,`' non ha significato di operatore.

2.8.7. Aritmetic Exceptions

A volte può accadere che alcune operazioni producano come risultato dei valori che non possono essere rappresentati con il tipo richiesto. In questo caso si ha una *aritmetic exception* chiamata di solito *overflow*. A volte si usa anche il termine *underflow* per indicare esplicitamente che il valore è troppo piccolo

Il generale il linguaggio C non specifica come le situazioni di overflow debbano essere trattate. In alcuni casi, i peggiori, viene semplicemente utilizzato un valore errato che può risultare sia dal troncamento della rappresentazione che da come le operazioni sono effettivamente valutate a livello di linguaggio macchina. In altri casi l'esecuzione del programma viene terminata con o senza un messaggio di errore. Infine può anche accadere che il sistema preveda metodi specifici per trattare le situazioni di aritmetic exception, ma questo dipende fortemente dal computer e dal compilatore.

Il linguaggio C è molto specifico su come vada trattato l'overflow solo nel caso in cui si tratti di interi senza segno. Infatti in questo caso lo Standard C richiede che tutte le operazioni siano fatte in modulo 2^n, dove n è il numero di bits utilizzati per la rappresentazione dell'intero

senza segno. Questo assicura che in ogni caso gli n bits meno significativi, ossia più a destra, siano sempre valutati correttamente.

In alcuni casi se si verificano certe condizioni sugli operandi (o argomenti delle funzioni) il linguaggio C dichiara esplicitamente che il risultato di alcune operazioni (o funzioni) è indeterminato, e quindi arbitrario.

Un sintomo tipico della presenza di aritmetic exceptions è il fatto che i risultati possano essere diversi a seconda del computer o del compilatore utilizzato, o che eseguendo il programma più volte i risultati cambino.

2.9. Espressioni logiche e controllo di flusso (Rev. 2.1.1)

Spesso nell'esecuzione di un programma può essere utile, se non necessario, dover operare delle scelte e prendere decisioni. Ad esempio se volessimo scrivere un programma che calcola le radici di un'equazione algebrica di secondo grado dobbiamo poter distinguere il caso con determinate negativo da quello positivo ed agire di conseguenza. Nel linguaggio C questo viene realizzato mediante l'istruzione condizionale "`if`", con o senza l'alternativa "else", che permette di eseguire una parte od un'altra del programma in base al risultato di una espressione logica.

2.9.1. Operatori di relazione e di uguaglianza

La forma più semplice di una espressione logica è

```
expr_1 op expr_2
```

dove "`expr_1`" e "`expr_2`" sono due espressioni e "`op`" un operatore binario di relazione o di uguaglianza:

Relazione	Test	Uguaglianza	Test
`<`	è minore di	`==`	è uguale a
`>`	è maggiore di	`!=`	è diverso da
`<=`	è minore o uguale di		
`>=`	è maggiore o uguale di		

Tutti questi operatori confrontano tra loro i propri operandi, per cui ad esempio

```
a < b
```

confronta se il valore di `a` è minore di quello `b`, mentre

```
a == b
```

confronta se i valori di `a` e `b` sono uguali.

I due operandi possono essere sia di tipo intero che floating-point. Nel C99 gli operandi degli operatori di uguaglianza possono anche essere di tipo complesso. Indipendentemente dal tipo degli operandi il valore di un operatore di relazione o di uguaglianza è sempre di tipo `int` ed è `0` se il test fallisce o `1` se il test è soddisfatto. Per cui, ad esempio,

```
int a = 1, b = 2;
printf("%d\n", a < b);
```

stampa `1`, mentre

```
int a = 1, b = 2;
printf("%d\n", a == b);
```

stampa `0`.

Gli operatori di relazione hanno una precedenza più alta degli operatori di uguaglianza, l'associatività è invece per tutti da sinistra.

È importante ricordare gli operatori operano sul valore degli operandi e che le espressioni sono interpretate secondo le regole di precedenza e di associatività. Questo vuol dire ad esempio che sebbene sia possibile scrivere un'espressione del tipo

```
1 < a < 8
```

questa ha un significato diverso dall'usuale notazione matematica. Infatti questa espressione viene interpretata come

```
((1 < a) < 8)
```

il cui risultato è ben diverso da quello voluto. Infatti siccome il valore di `(1 < a)` è o `0` o `1`, ed entrambi sono minori di 8, il valore dell'espressione logica "`1 < a < 8`" è sempre `1` indipendentemente dal valore di `a`! È possibile scrivere espressioni logiche che abbiano lo stesso significato dell'usuale notazione matematica unendo più espressioni logiche semplici mediante gli operatori logici.

2.9.2. Operatori logici

Il C fornisce i due operatori logici binari `AND` e `OR` e l'operatore logico unario `NOT`

Operatore	Operazione
`!`	`NOT`
`&&`	`AND`
`||`	`OR`

Il livello di precedenza dei tre operatori è decrescente con l'operatore "`!`" che ha la precedenza più alta e l'operatore "`||`" più bassa. Anche le regole di associatività sono differenti, da destra per l'operatore "`!`" e da sinistra per gli operatori "`&&`" e "`||`".

Come per gli operatori di relazione e di uguaglianza anche per gli operatori logici il valore è

sempre di tipo `int` ed è `0` se l'asserzione logica è *falsa* e `1` se invece è *vera*. In particolare il valore dell'operatore `NOT` "`!`" è `1` se il valore dell'operando a cui è applicato è uguale a `zero` nel senso dell'operatore "`==`", e `0` nel caso contrario. Per cui l'espressione `!(a)` ha lo stesso significato di `(a) == 0`.

Gli operatori logici binari `AND` e `OR` sono anche chiamati `AND` *condizionale* e `OR` *condizionale* poiché l'operando di destra può non essere valutato se il valore dell'operando di sinistra è sufficiente per determinare la veridicità o no dell'asserzione logica:

- `AND`:
  L'operando di sinistra viene valutato e se il suo valore è uguale a zero nel senso dell'operatore "`==`" l'operando di destra non viene valutato ed il risultato è `0`. In caso contrario, ossia se il valore dell'operando di sinistra è diverso da zero, l'operando di destra viene valutato ed il risultato è `0` o `1` a seconda che il valore del secondo operando sia uguale o diverso zero nel senso dell'operatore "`==`". I possibili casi sono riassunti qui di seguito

AND: `expr_1`	`expr_2`	`expr_1 && expr_2`
`== 0`	non valutata	`0`
`!= 0`	`== 0`	`0`
`!= 0`	`!= 0`	`1`

- `OR`:
  L'operando di sinistra viene valutato e se il suo valore è diverso da zero nel senso dell'operatore "`!=`" l'operando di destra non viene valutato ed il risultato è `1`. In caso contrario, ossia se il valore dell'operando di sinistra è uguale a zero, l'operando di destra viene valutato ed il risultato è `0` o `1` a seconda che il valore del secondo operando sia uguale o diverso zero nel senso dell'operatore "`==`". I possibili casi sono riassunti qui di seguito

OR: `expr_1`	`expr_2`	`expr_1 || expr_2`
`!= 0`	non valutata	`1`
`== 0`	`== 0`	`0`
`== 0`	`!= 0`	`1`

Per illustrare queste regole supponiamo che "`a = 0`" e "`b = 1`" allora

`a && b`	$\Rightarrow$	vale `0` e `b` non è valutato
`b && a`	$\Rightarrow$	vale `0` e `a` è valutato
`a || b`	$\Rightarrow$	vale `1` e `b` è valutato
`b || a`	$\Rightarrow$	vale `1` e `a` non è valutato

Questa proprietà degli operatori logici fa si che in alcuni casi la correttezza o no del programma dipenda dall'ordine con cui sono scritti i due operandi. Incontreremo esempi più avanti.

Utilizzando gli operatori logici è possibile concatenare più espressioni logiche. Ad esempio la condizione matematica `1 < a < 8` può essere scritta come

```
1 < a  &&  a < 8
```

Le parentesi non sono necessarie poiché gli operatori di relazione hanno un livello di precedenza più alto, per cui questa espressione viene correttamente valutata come

```
(1 < a)  &&  (a < 8)
```

In caso di indecisione è buona norma utilizzare le parentesi, anche se non strettamente necessarie. Inoltre un moderato uso delle parentesi può migliorare la comprensione del programma: la seconda scrittura è molto più chiara della prima.

2.9.3. Istruzione condizionale `if`

L'istruzione condizionale "`if`" permette di eseguire o non eseguire una o più istruzioni a seconda che una condizione logica sia verificata o no. La forma generale di un'istruzione condizionale è

```
if (expression)
  statement;
```

dove `expression` è l'*espressione di controllo* dell'istruzione e `statement` il *corpo*. L'istruzione condizionale viene valutata calcolando per prima cosa il valore dell'espressione di controllo `expression` che può essere sia di tipo numerico che logico. Se il suo valore è diverso da zero nel senso "`!= 0`" la condizione è considerata vera e `statement` viene valutato. Nel caso contrario, ossia se il valore di `expression` è uguale a `0` nel senso "`== 0`" `statement` viene ignorato. Ad esempio le seguenti istruzioni

```
if (i < 0) printf("i e' negativo!\n");
```

scrivono un messaggio ogni qual volta il valore della variabile `i` è minore di zero.

È possibile eseguire più istruzioni soggette ad una condizione raggruppandole insieme con una coppia di parentesi graffe "{}":

```
if (expression)
 {
   statement_1;
   statement_2;
   .....
   statement_n;
 }
```

In questo caso tutto il blocco di istruzioni racchiuse dalle parentesi graffe forma il corpo dell'istruzione `if` che viene quindi eseguito od ignorato a seconda del valore dell'espressione di controllo `expression`.

Un gruppo di istruzioni raggruppato da parentesi graffe viene chiamato *istruzione composta* per cui possiamo dire che il corpo dell'istruzione `if` può essere formato sia una sola istruzione che da un'istruzione composta.

Un'istruzione composta può contenere sia istruzioni che dichiarazioni. Nel C99 le istruzioni e le dichiarazioni possono essere mischiate, nel C89 tutte le dichiarazioni devono obbligatoriamente precedere le istruzioni.

Nota di stile. Per maggior chiarezza le istruzioni racchiuse dalle parentesi "{}" sono usualmente *indentate* in modo da rendere più facile l'identificazione delle istruzioni escluse quando l'espressione di controllo logica è falsa. Una corretta indentazione è alla base della scrittura di programmi comprensibili.

2.9.4. Costruzione `if-else`

Alternative possono essere introdotte nell'istruzione condizionale con l'alternativa "`else`". La forma di un'istruzione condizionale con alternativa è

```
if (expression)
  statement_true
else
  statement_false
```

dove sia `statement_true` che `statement_false` possono essere formati sia da una sola istruzione che da un'istruzione composta. L'istruzione condizionale viene eseguita valutando per prima cosa l'espressione di controllo `expression` e se il suo valore è differente da zero nel senso "`!= 0`" (condizione *vera*) viene eseguito `statement_true`. Se invece il suo valore è zero nel senso "`== 0`" (condizione *falsa*) viene eseguito `statement_false`. Ad esempio le seguenti istruzioni

```
if (i < 0)
  printf("i e' negativo!\n");
else
  printf("i e' non negativo!\n");
```

stampano un messaggio diverso a seconda del segno della variabile `i`.

Una istruzione condizionale può contenere al suo interno altre istruzioni condizionali con o senza l'alternativa "`else`". Questa libertà però può dare origine ad una ambiguità nota come il *Dangling-Else Problem*. Supponiamo ad esempio di voler sapere se un numero intero di valore inferiore a 20 sia pari o dispari. Questo può essere ottenuto con le istruzioni seguenti

```
if (i < 20)
   if ( (i % 2) == 0)
     printf("i e' pari!\n");
else
   printf("i e' dispari!\n");
```

In questo esempio vi sono due `if` ed una sola alternativa `else`. Le istruzioni sono sintatticamente corrette ma a quale `if` l'altrenativa `else` appartiene? Non vi è modo di risolvere l'ambiguità e decidere a quale `if` è associata l'alternativa `else`. Lo Standard C risolve

l'ambiguità richiedendo arbitrariamente che l'alternativa `else` sia associata all'istruzione `if` più interna, nel nostro esempio la seconda per cui le istruzioni fanno quanto voluto.

Questa ambiguità può essere superata utilizzando le parentesi graffe per racchiudere le istruzioni controllate dall'istruzione `if`. Ad esempio le precedenti istruzioni possono essere scritte come

```
if (i < 20) {
   if ( (i % 2) == 0)
      printf("i e' pari!\n");
   else
      printf("i e' dispari!\n");
}
```

Nota di stile. Osserviamo che questa strategia oltre a risolvere il problema del dangling-else migliora la lettura del programma, con tutti i vantaggi che ne conseguono. Chiaramente non bisogna eccedere nell'altro senso, infatti un eccessivo uso di parentesi può rendere difficoltosa la lettura al pari della loro mancanza. Un buon compromesso è quello di utilizzare le parentesi solo se le istruzioni controllate dall'istruzione `if` sono più di una istruzione, come nell'esempio precedente.

Dal momento che l'alternativa `else` nello Standard C viene associata all'istruzione `if` più interna, scelte multiple possono essere facilmente realizzate con una serie di istruzioni `if-else` in cascata in cui tutte le istruzioni `if`, esclusa l'ultima, hanno un'altra istruzione `if` alla corrispondente alternativa `else`:

```
if (expression_1)
   statement_1
else if (expression_2)
   statement_2
else if (expression_3)
   statement_3
...
else
   statement_n
```

Infatti indentando le istruzioni in modo da incolonnare ogni istruzione `if` con la corrispondente alternativa `else` si ha

```
if (expression_1)
   statement_1
else if (expression_2)
        statement_2
     else if (expression_3)
             statement_3
          else if (expression_4)
                  statement_4
                       ...
                  else
                     statement_n
```

da cui la struttura risulta chiara. È evidente che una corretta indentatura può essere fondamentale nella scrittura delle istruzioni condizionate.

2.9.5. Espressioni condizionate

Scelte possono essere effettuate anche utilizzando gli operatori "`?`" e "`:`" che introducono una espressione condizionata della forma

```
expr_1 ? expr_2 : expr_3
```

L'operatore "`?:`" è un operatore ternario poiché prende tre operandi. L'espressione condizionata viene valutata calcolando il valore del primo operando `expr_1` che può essere sia di tipo numerico che logico. Se il valore è zero nel senso "`== 0`" (condizione *falsa*) allora viene valutato il secondo operando `expr_2` ed il suo valore diventa il valore dell'espressione. Il terzo operando non viene valutato. Se invece il valore del primo operando è diverso da zero nel senso "`!= 0`" (condizione *vera*) viene valutato il terzo operando `expr_3` ed il suo valore diventa il valore dell'espressione. In questo caso è il secondo operando a non essere valutato. In ogni caso solo una delle due espressioni `expr_2` e `expr_3` viene valutata.

L'espressione condizionata è equivalente alla costruzione `if-else`

```
if (expr_1)
      value = expr_2;
else
      value = expr_3;
```

dove `value` è il valore dell'espressione condizionata, ma ha in vantaggio di poter essere usata nelle espressioni. Supponiamo ad esempio di voler stampare il valore del minimo dei due numeri `x` e `y`. Questo può essere ottenuto utilizzando la costruzione `if-else` come

```
if (x < y) {
   printf("minimum: %d\n", x);
}
else {
   printf("minimum: %d\n", y);
}
```

oppure in forma più elegante utilizzando un'espressione condizionata come

```
printf("minimum: %d\n", (x < y) ? x : y);
```

Le parentesi non sono strettamente necessarie ma sono utili per evidenziare la prima espressione dell'espressione condizionata.

Il livello di precedenza dell'espressione condizionata è inferiore a quello degli operatori logici mentre la sua associatività rispetto al primo e terzo operando e da destra. Questo vuol dire che ad esempio l'espressione

```
a ? b : c ? d : e ? f : g
```

viene valutata come

```
a ? b : (c ? d : (e ? f : g))
```

In questo caso tuttavia l'uso delle parentesi, anche se non richieste, è consigliabile.

Nota di stile. Per una migliore lettura del programma si suggerisce di racchiudere sempre il primo operando tra parentesi tonde "`()`", anche se questo non è obbligatorio.

2.9.6. Istruzione `switch`

La forma generale dell'istruzione `switch` è

```
switch (expression) {
  case constant_1:
                  statement_1
  ........
  case constant_n:
                  statement_n
  default:
                  statement_d
}
```

dove `expression` è un'espressione di tipo intero e `constant_1`, ..., `constant_n` sono delle costanti di tipo intero. Il corpo dell'istruzione è composto da zero o più *labels* `case` individuate dal valore della costante corrispondente, e da una *label opzionale* `default`.

L'istruzione `switch` viene eseguita valutando l'espressione di controllo `expression` e confrontando il suo valore con quello delle costanti intere delle varie labels `case`. Se il valore, eventualmente convertito in tipo intero, è uguale al valore di una delle costanti intere che identificano le labels l'esecuzione del programma continua dalla label `case` corrispondente. Se invece il valore dell'espressione di controllo non è uguale a nessuna delle costanti intere ma vi è la label `default` il programma continua dal punto indicato dalla label `default` altrimenti, ossia se manca la label `default`, nessuna delle istruzioni del corpo viene eseguita ed il programma continua con quanto segue l'istruzione `switch`.

L'istruzione `switch` ricorda molto la costruzione con una serie di istruzioni `if-else`, tuttavia vi sono differenze. Per prima cosa l'ordine con cui il valore dell'espressione di controllo viene confrontato con quello delle costanti non è necessariamente l'ordine con cui sono scritte le labels `case`. Questo in generale dipende dal sistema ed anche da numero di labels presenti. La seconda differenza è che quando il flusso del programma viene trasferito ad una label `case` o `default` l'esecuzione del programma continua a partire da quel punto ignorando tutte le eventuali labels successive, e quindi eseguendo tutte le istruzioni fino alla fine del corpo. Il seguente programma illustra questo punto.

Programma: `switch_1.c`

```
#include <stdio.h>

int main(void)
{
    int var;
```

```
    printf("valore intero : ");
    scanf("%d", &var);

    switch (var) {
       case 1:
         printf("var vale 1\n");
       case 2:
         printf("var vale 2\n");
       case 3:
         printf("var vale 3\n");
       default:
         printf("non valido\n");
    }

    return 0;
}
```

Se al programma viene fornito il valore 1 il controllo passa alla prima label `case` per cui il risultato è

```
valore intero : 1
var vale 1
var vale 2
var vale 3
non valido
```

in quanto tutte le istruzioni del corpo sono eseguite. Se invece viene digitato il valore 3 il controllo viene trasferito all'ultima label `case` e saranno eseguite tutte le istruzioni da questo punto fino alla fine del corpo. Di conseguenza in output si avrà

```
valore intero : 3
var vale 3
non valido
```

Se si desidera che l'esecuzione del corpo dell'istruzione `switch` termini dopo l'esecuzione di una singola label, o in un punto ben determinato, si deve utilizzare l'istruzione `break`:

Programma: `switch_2.c`

```
#include <stdio.h>

int main(void)
{
    int var;

    printf("valore intero : ");
    scanf("%d", &var);

    switch (var) {
       case 1:
         printf("var vale 1\n");
         break;
```

```
        case 2:
          printf("var vale 2\n");
          break;
        case 3:
          printf("var vale 3\n");
          break;
        default:
          printf("non valido\n");
          break;
    }

    return 0;
}
```

Adesso il programma scrive correttamente il valore della variabile `var`:

```
valore intero : 1
var vale 1
```

ovvero

```
valore intero : 3
var vale 3
```

Osserviamo che nell'ultima label, la label `default` nell'esempio, l'istruzione `break` non è strettamente necessaria, è tuttavia una buona prassi quella di includerla per evitare che un'eventuale aggiunta di un'altra label alla fine possa creare problemi.

Non vi è un ordine preciso con cui debbano essere scritte le labels `case` e `default`. Queste possono essere messe in un punto qualsiasi del corpo dell'istruzione `switch`, anche all'interno di altre istruzioni. Le uniche restrizioni sono sul numero di labels `case` che possono comparire nel corpo di una stessa istruzione `switch`, nel C89 il limite è 257 mentre il C99 ne prevede fino a 1023, e che le costanti intere che identificano le labels `case` siano diverse tra loro.

Se non scritte in modo ordinato le istruzioni `switch` possono produrre programmi di difficile lettura, come mostra il seguente programma che determina se un numero intero tra 0 e 10 (esclusi) è pari o dispari:

Programma: `bad_switch.c`

```
#include <stdio.h>

int main(void)
{
    int var;

    printf("valore intero : ");
    scanf("%d", &var);

    switch (var) {
       default:
         if (var > 0 && var < 10 ) {
```

```
            case 2: case 4: case 6: case 8:
              printf("numero pari\n");
              break;
            case 1: case 3: case 5: case 7:
              printf("numero dispari\n");
          }
          else
            printf("non valido\n");
    }

    return 0;
}
```

Per limitare i problemi che potrebbero derivare da un uso non corretto dell'istruzione `switch` suggeriamo di utilizzare sempre la seguente regola di stile:

- Scrivere sempre le labels come istruzioni principali all'interno del corpo e mai all'interno di altre istruzioni, come nell'esempio precedente;
- Terminare ogni label con l'istruzione `break` o con un commento che indichi che l'esecuzione del programma continua con la prossima label.

È possibile interrompere l'esecuzione del corpo dell'istruzione `switch` anche con le istruzioni `return`, `goto` e `continue` che trasferiscono il flusso del programma ad altre parti del programma. Vedremo queste istruzioni più avanti nel corso di questo primer.

2.9.7. Istruzione `goto`

Il linguaggio C fornisce anche l'istruzione `goto` per trasferire il controllo del programma ad un altro punto del programma purché all'interno della stessa funzione.

La sintassi dell'istruzione `goto` è

```
goto name_label;
```

dove `name_label` è una *label* all'interno della stessa funzione.

Il seguente programma che determina se un numero intero compreso tra 0 e 10 è pari o dispari illustra l'uso dell'istruzione `goto`

Programma: `goto.c`

```
#include <stdio.h>

int main(void)
{
    int var;

    printf("valore intero : ");
    scanf("%d", &var);
```

```
    if (var <= 0 ||  var >= 10 ) goto fine;

    if ( var % 2 == 0 )
            printf("numero pari\n");
    else
      printf("numero dispari\n");

    return 0;

 fine:
    printf("non valido\n");

    return 1;
}
```

Non diremo molto su questa istruzione perché il suo uso è considerato un pessimo stile di programmazione non solo perché spesso rende difficile l'ottimizzazione del programma da parte del compilatore, ma sopratutto perché un programma con "salti" è difficile da leggere e da mantenere.

2.10. Cicli (Rev. 2.1)

Supponiamo di voler scrivere un programma che calcoli la somma dei primi 5 numeri interi. Una possibile strategia potrebbe essere

```
int sum;
sum = 1 + 2 + 3 + 4 + 5;
```

La strategia non è sbagliata ma chiaramente diventa inattuabile quando i numeri da sommare aumentano ad esempio 50, 100 o più. Per individuare una strategia o *algoritmo* migliore riscriviamo le precedenti istruzioni come

```
int sum;
sum  = 0;
sum += 1;
sum += 2;
sum += 3;
sum += 4;
sum += 5;
```

Il risultato è chiaramente lo stesso. Tuttavia scritte in questo modo risulta evidente che per eseguire la somma voluta si deve ripetere la stessa operazione più volte. Situazioni di questo genere in cui la stessa operazione deve essere ripetuta più volte sono piuttosto frequenti per cui è utile avere a disposizione istruzioni di controllo che permettano di eseguire ripetutamente un certo numero di istruzioni. Queste sono chiamate istruzioni di iterazione o di ciclo. Il linguaggio C fornisce tre diverse istruzioni di ciclo.

2.10.1. Istruzione `while`

La forma generale del dell'istruzione `while` è

```
while (expression)
    statement
```

dove `expression` è l'*espressione di controllo* del ciclo e `statement` il *corpo* del ciclo. Il corpo può essere formato sia una sola istruzione che da un'istruzione composta, ossia da definizioni ed istruzioni raggruppate da una coppia di parentesi graffe "{}".

L'istruzione `while` viene eseguita valutando per prima cosa l'espressione di controllo. Se il valore dell'espressione `expression` è diverso da zero (condizione *true*) viene eseguito `statement`. L'intera procedura viene ripetuta valutando alternativamente `expression` e poi, se il suo valore è diverso da zero, `statement` fino a quando il valore di `expression` non diventa uguale a `0` (condizione *false*). Il valore di `expression` può variare sia per effetto di istruzioni in `statement` che in `expression`.

L'istruzione `while` controlla la condizione di controllo del ciclo prima di eseguire le istruzioni che compongono `statement`, di conseguenza se la prima volta che `expression` viene valutata il suo valore è zero, e quindi la condizione è *falsa*, il corpo del ciclo `while` non viene mai eseguito.

Utilizzando l'istruzione `while` è facile scrivere il programma che calcola la somma dei primi 5 numeri interi. È infatti un semplice esercizio verificare che questo può essere scritto come:

```
int sum;
int num;

num = 1;                /* primo numero                    */
sum = 0;                /* all'inizio la somma e' 0        */
while ( num <= 5 ) {    /* somma tutti i numeri fino a 5 */
  sum += num;           /* si aggiunge il nuovo numero    */
  ++num;                /* prossimo numero                 */
}
```

Il vantaggio di questa scrittura è evidente, infatti se adesso volessimo calcolare la somma dei numeri da 1 a 100 basta modificare una sola linea e non tutto il programma.

Come ulteriore esempio consideriamo il seguente programma che calcola la sequenza dei numeri di Fibonacci

$$1\ 1\ 2\ 3\ 5\ 8\ 13\ \ldots$$

minori di 100. I numeri di Fibonacci sono ottenuti facilmente con la formula di ricorrenza

$$f_{n+1} = f_n + f_{n-1}, \qquad n > 1$$

con $f_0 = f_1 = 1$, che mostra che ogni numero della sequenza è dato dalla somma dei due numeri precedenti.

Programma: `fibonacci_1.c`

```
/*
 * Descrizione :  Calcola i numeri di Fibonacci minori di 100
 *
 * $yaC-Primer: fibonacci_1.c, v 1.1 27.02.2005 AC $
 */
#include <stdio.h>

int main(void)
{
  int f_nm1, f_n, f_np1;      /* f_(n-1), f_n, f_(n+1) */

  f_nm1 = 1;                  /* i primi due numeri sono 1 e 1 */
  f_n   = 1;

  printf("%d\n", f_n);        /* stampa primo numero */

  while (f_n < 100) {
    printf("%d\n", f_n);

    f_np1 = f_n + f_nm1;        /* f_(n+1) = f_n + f_(n-1) */

    /* n -> n+1 */
    f_nm1 = f_n;               /* f_(n+1) -> f_n      */
    f_n   = f_np1;             /* f_n     -> f_(n-1) */
  }
  return 0;
}
```

In questo esempio, come nel precedente, le istruzioni che modificano il valore della condizione di controllo e ed eventualmente interrompono il ciclo, sono nel corpo dell'istruzione `while`. Questo non è tuttavia obbligatorio ed infatti queste possono trovarsi direttamente nell'espressione della condizione di controllo, come mostra il seguente esempio. Sfruttando il fatto che nel linguaggio C il valore dell'operatore di assegnamento "`=`" è il valore assegnato il programma per i numeri di Fibonacci può anche essere scritto come

Programma: `fibonacci_2.c`

```
/*
 * Descrizione :  Calcola i numeri di Fibonacci minori di 100
 *
 * $yaC-Primer: fibonacci_2.c, v 1.1 27.02.2005 AC $
 */
#include <stdio.h>

int main(void)
{
  int f_nm1, f_n, f_np1;      /* f_(n-1), f_n, f_(n+1) */

  f_nm1 = 0;                  /* f_(-1) = 0 */
  f_n   = 1;                  /* f_1    = 1 */
```

```
  printf("%d\n", f_n);        /* stampa primo numero */

  while ((f_np1 = f_n + f_nm1) < 100) {
    printf("%d\n", f_np1);

    f_nm1 = f_n;                /* f_n      -> f_(n-1) */
    f_n   = f_np1;              /* f_(n+1)  -> f_n     */
  }
  return 0;
}
```

Entrambi i programmi producono il risultato voluto, tuttavia è evidente che il primo è molto più facile da leggere e capire. Di conseguenza, utilizzando il principio che tanto più i programmi sono semplici da capire tanto più sono facili da mantenere e da controllare, la prima versione è preferibile alla seconda.

2.10.2. Istruzione `for`

La forma generale dell'istruzione `for` è

```
for (initial_expr; control_expr; iteration_expr)
    statement
```

Le tre espressioni nelle parentesi sono opzionali ed una o più di esse, o anche tutte, possono essere omesse. I punti e virgola "`;`" che separano le tre espressioni sono invece obbligatori e non possono essere omessi anche se vengono omesse una o più espressioni. Sia `initial_expr` che `iteration_expr` possono essere formate da espressioni sequenziali, ossia più espressioni separate dall'operatore "`,`". Nel C99 `initial_exp` può essere sostituita da una dichiarazione che dichiara ed inizializza le variabili di controllo utilizzate nel ciclo. Questo permette in genere una migliore ottimizzazione del ciclo da parte del compilatore. Infine `statement`, il corpo del ciclo, può essere formato da una sola istruzione o da un'istruzione composta.

L'istruzione `for` viene eseguita nel modo seguente:

1. Se presente viene valutata l'espressione `inital_expr`. Il suo eventuale valore viene trascurato.
2. Se presente viene valutata l'espressione `control_expr` ed utilizzata come condizione di controllo. Se il suo valore è zero il ciclo è completo ed l'esecuzione del programma continua con le istruzioni che seguono il corpo del ciclo. Se invece il valore non è nullo, o se `control_expr` manca, la condizione è considerata *true* e si passa al prossimo punto.
3. Viene eseguito il corpo `statement` del ciclo `for`.
4. Se presente viene valutata l'espressione `iteration_expr`. Il suo eventuale valore viene trascurato.
5. Si ritorna al punto 2.

. Il ciclo `for` è quindi formalmente equivalente alla seguente istruzione `while`

```
initial_expr;
while (control_expr) {
    statement;
    iteration_expr;
}
```

L'espressione `initial_expr` viene usualmente chiamata "condizione di inizializzazione" perché viene normalmente utilizzata per inizializzare le variabili che controllano il ciclo, ossia quelle usate nell'espressione di controllo `control_expr` ed il cui valore viene utilizzato per determinare se il ciclo deve continuare o no. Infine la terza espressione `iteration_expr` viene normalmente utilizzata per modificare le variabili che controllano l'esecuzione del ciclo, questa tuttavia può anche essere utilizzata per eseguire istruzioni non necessariamente legate alla modifica di quest'ultime.

Come avviene per il istruzione `while` anche nel caso dell'istruzione `for` se l'espressione di controllo è falsa all'inizio del ciclo il corpo `statement` non viene mai eseguito.

Il problema di sommare i primi cinque numeri interi positivi può essere risolto anche con un'istruzione `for` come

```
int sum;
int num;

sum = 0;
for (num = 1; num <= 5; ++num) {
   sum += num;
}
```

Le espressioni `initial_expr` e `iteration_expr` possono essere composte da più espressioni separate dall'operatore virgola "`,`" per cui è anche possibile scrivere

```
int sum;
int num;

for (sum = 0, num = 1; num <= 5; ++num) {
   sum += num;
}
```

in modo da raggruppare insieme le inizializzazioni delle variabili usate nel ciclo.

È possibile spingere il raggruppamento oltre e scrivere

```
int sum;
int num;

for (sum = 0, num = 1; num <= 5; sum += num, ++num);
```

raggruppando tutte le istruzioni del ciclo nelle espressioni `initial_expr` e `iteration_expr` e lasciando così `statement`, ossia il corpo dell'istruzione `for`, composto dalla sola istruzione vuota (*null*) data dal solo punto e virgola "`;`". Sebbene sia sempre possibile ridurre il corpo dell'istruzione `for` alla sola istruzione vuota, uno stile di programmazione che utilizza questa forma in modo sistematico è chiaramente uno stile piuttosto questionabile.

Infine osserviamo che siccome nell'istruzione `for` una o più delle espressioni fra le parentesi "`()`" possono essere omesse, le seguenti istruzioni

```
int sum;
int num;

sum = 0;
num = 1;
for ( ; num <= 5; ++num) {
   sum += num;
}
```

sono perfettamente lecite ed equivalenti alle forme precedenti.

A volte può essere utile avere più di una variabile che controlla il ciclo `for`. In questi casi l'operatore "`,`" risulta molto utile perché permette di raggruppare più istruzioni di assegnamento in una sola istruzione, come mostra il seguente frammento di programma

```
int i, j;
int a[10], b[20];

for (i = 0, j = 0; i < 10 && j < 20; i += 1; j += 2)
  a[i] = b[j];
```

che copia una parte dell'array `b` nell'array `a`.

2.10.3. Istruzione `do`

L'istruzione `do` è molto simile all'istruzione `while` ma differisce da quest'ultima, e dall'istruzione `for`, per il fatto che il corpo del ciclo viene eseguito almeno una volta indipendentemente dal valore dell'espressione di controllo.

La forma generale dell'istruzione `do` è

```
do
  statement
while (expression);
```

Nell'istruzione `do` viene eseguito per primo `statement` e poi valutata l'espressione di controllo `expression`. Se il risultato di quest'ultima è *true* (valore non nullo) allora l'intera procedura viene ripetuta eseguendo alternativamente `statement` e valutando l'espressione di controllo `expression` fino a quando il risultato dell'espressione di controllo non diventa *false* (valore nullo) e il ciclo termina.

Il valore dell'espressione di controllo `expression` può essere modificato con istruzioni sia all'interno del corpo `statement` del ciclo che nell'espressione di controllo stessa.

È possibile scrivere un ciclo `do` il cui corpo è composto dalla sola istruzione vuota "`;`"

```
do ;
  while (expression);
```

Questo ciclo viene, tuttavia, usualmente scritto utilizzando l'istruzione `while`:

```
while (expression);
```

Siccome nel ciclo `do` il corpo del ciclo viene sempre eseguito almeno una volta l'istruzione `do` è usata più raramente delle istruzioni `while` e `for` nella costruzione dei cicli.

2.10.4. Istruzioni `break` e `continue`

L'esecuzione di un ciclo può essere modificata utilizzando le istruzioni `break` e `continue`. L'utilizzo di queste istruzioni al di fuori del corpo di un ciclo è illegale.

Istruzione `break`: L'istruzione `break` interrompe la normale esecuzione del ciclo `while`, `do` o `for` più interno cosicché le istruzioni del ciclo che seguono l'istruzione `break` non vengono eseguite e l'esecuzione del programma continua con le istruzioni immediatamente successive al ciclo.

Supponiamo ad esempio di voler scrivere un programma che calcoli la somma di una serie di numeri interi letti da terminale di lunghezza arbitraria. Questo può essere realizzato con le seguenti istruzioni:

```
sum = 0;
while (1) {
  printf("numero o 0 per terminare: ");
  scanf("%d", &number);
  if (number == 0) break;
  sum += number;
}
printf("\nTotale: %d\n", sum);
```

Per poter leggere una sequenza di lunghezza arbitraria il ciclo inizia con

```
while (1) {
```

per cui la condizione è sempre *true*. Chiaramente in questo modo il ciclo non terminerebbe mai (ciclo infinito), per cui l'unico modo di uscirne è quella di interrompere il ciclo da qualche parte. L'esecuzione del ciclo viene controllata dall'istruzione

```
if (number == 0) break;
```

cosicché i valori letti sono sommati fino quando non viene inserito il valore 0. Quando viene letto il valore 0 l'istruzione `break` interrompe il ciclo per cui le istruzioni seguenti del ciclo vengono eseguite ed il programma continua con l'istruzione

```
printf("\nTotale: %d\n", sum);
```

che scrive sul terminale la somma dei numeri letti.

Anche l'istruzione

```
for( ; ; ) {
...
}
```

genera un ciclo infinito che va quindi interrotto con un'istruzione `break` da qualche parte.

Istruzione `continue`: L'istruzione `continue`, come l'istruzione `break`, modifica l'esecuzione di un ciclo. La differenza è che mentre l'istruzione `break` termina il ciclo e l'esecuzione del programma continua con le istruzioni immediatamente successive al ciclo, l'istruzione `continue` interrompe la normale esecuzione dell'iterazione del ciclo più interno e l'esecuzione salta all'iterazione successiva del ciclo trascurando le istruzioni successive del ciclo. Questo vuol dire che quando viene incontrata l'istruzione `continue` il ciclo salta direttamente alla valutazione dell'espressione di controllo del ciclo, valutando l'espressione di incremento per il ciclo `for`, e nel caso il risultato sia `true` il ciclo continua con l'esecuzione del corpo del ciclo.

Per illustrare la differenza tra le due istruzioni supponiamo che nel programma che calcola la somma dei numeri immessi da terminale si vogliano sommare solo i numeri positivi. Questo si ottiene facilmente con l'istruzione `continue`

```
sum = 0;
while (1) {
  printf("numero o 0 per terminare: ");
  scanf("%d", &number);
  if (number == 0) break;
  if (number < 0) continue;
  sum += number;
}
printf("\nTotale: %d\n", sum);
```

In questo modo infatti se viene letto un numero negativo le istruzioni seguenti del ciclo, nel caso specifico la somma, non vengono eseguite ed il ciclo inizia nuovamente dall'inizio chiedendo un nuovo numero.

Come l'istruzione `break` anche l'istruzione `continue` può essere usata con tutte e tre le istruzioni `while`, `for` e `do`. I seguenti programmi mostrano l'ordine con cui le diverse parti di un ciclo sono eseguite quando si incontrano le istruzioni `break` e `continue`. I programmi utilizzano la funzione

```
#include <stdio.h>

int putchar(int c);
```

per scrivere un carattere che identifichi delle diverse parti del corpo del ciclo.

Programma: `while-continue.c`

```
#include <stdio.h>

int main(void)
{
  int i = 0;

  while(putchar('1')) {
    putchar('2');
    if (++i == 2) {
      putchar('b');
      break;
    }
```

```
    putchar('c');
    continue;
    putchar('3');
  }
  putchar('4');
  printf("\n");
  return 0;
}
```

Quando questo programma viene eseguito si ha la sequenza di caratteri

```
12c12b4
```

che mostra chiaramente che nel caso dell'istruzione `while` ogni volta che viene incontrata l'istruzione `continue`, carattere '`c`', il ciclo inizia da capo valutando per prima cosa l'espressione di controllo, carattere '`1`'. L'istruzione `break`, carattere '`b`', interrompe invece il ciclo e viene eseguita la prima istruzione subito dopo il ciclo, carattere '`4`'.

Programma: `for-continue.c`

```
#include <stdio.h>

int main(void)
{
  int i = 0;

  for(putchar('1'); putchar('2'); putchar('3')) {
    putchar('4');
    if (++i == 2) {
      putchar('b');
      break;
    }
    putchar('c');
    continue;
    putchar('5');
  }
  putchar('6');
  printf("\n");
  return 0;
}
```

In questo caso si ha la sequenza di caratteri

```
124c324b6
```

per cui nel caso dell'istruzione `for` ogni volta che viene incontrata l'istruzione `continue`, carattere '`c`', il ciclo inizia da capo valutando per prima cosa l'espressione di iterazione, carattere '`3`' e poi quella di controllo, carattere '`2`'. L'istruzione `break`, carattere '`b`', interrompe invece il ciclo e viene eseguita la prima istruzione subito dopo il ciclo, carattere '`6`'.

Programma: `do-continue.c`

```c
#include <stdio.h>

int main(void)
{
  int i = 0;

  do{
    putchar('1');
    if (++i == 2) {
      putchar('b');
      break;
    }
    putchar('c');
    continue;
    putchar('2');
  } while(putchar('3'));
  putchar('4');
  printf("\n");
  return 0;
}
```

Infine nel caso dell'istruzione `do` si ha la sequenza di caratteri

```
1c31b4
```

da cui si vede ogni volta che viene incontrata l'istruzione `continue`, carattere '`c`', il ciclo inizia da capo valutando per prima cosa l'espressione di controllo, carattere '`3`'. Come nei casi precedenti l'istruzione `break`, carattere '`b`', interrompe il ciclo e viene eseguita la prima istruzione subito dopo il ciclo, carattere '`4`'.

In ogni caso le istruzioni del ciclo dopo l'istruzione `continue` non vengono mai eseguite.

2.11. Esempio: Rappresentazione binaria di un numero intero decimale di tipo unsigned (Rev. 2.1.1)

Come primo esempio di programmazione consideriamo un semplice programma che fornisce la rappresentazione binaria con un numero di bits fissato di un numero intero decimale di tipo `unsigned`. Vi sono vari modi di scrivere il programma, quello discusso qui di seguito utilizza le conoscenze del linguaggio C acquisite fino ad ora. Lo sviluppo e la struttura del programma saranno descritte in dettaglio e serviranno come base per tutti i programmi che saranno discussi nel seguito.

La scrittura di un programma segue uno schema ben preciso che è indipendente dal problema che si vuole risolvere, dal linguaggio di programmazione scelto e dal sistema utilizzato.

La prima cosa da fare quando ci si accinge a scrivere un programma è quella di individuare la procedura o *algoritmo* che permette di risolvere il problema mediante una serie di operazioni semplici. Queste sono poi trasformate in istruzioni per il computer utilizzando il linguaggio

di programmazione scelto, il linguaggio C nel nostro caso. L'insieme di tutte le istruzioni, scritte in uno o più files, formano il programma che risolve il problema.

È bene tener presente che non si può scrivere un programma se prima non si individua l'algoritmo. Un errore tipico dei programmatori alle prime armi è quello di iniziare a scrivere il programma senza avere bene in mente, o su un pezzo di carta, l'algoritmo da utilizzare.

L'algoritmo per determinare la rappresentazione binaria di un numero intero decimale di tipo `unsigned` è piuttosto semplice: basta dividere ripetutamente il numero per 2. La rappresentazione binaria del numero è data dal resto delle divisioni. Infatti il resto della prima divisione fornisce il primo bit, quello meno significativo, il resto della divisione della parte intera della prima divisione il secondo bit e così via. Ripetendo la procedura tante volte quanti sono i bits desiderati si ottiene la rappresentazione cercata.

Il seguente programma trasforma questo l'algoritmo in una serie di istruzioni in linguaggio C.

Programma: `unsigned2bin_0.c`

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale unsigned.
 *
 * $yaC-Primer: unsigned2bin_0.c v 1.0 27.01.05 AC $
 */
#include <stdio.h>

int main(void)
{
  int  number;               /* numero decimale positivo     */
  int  n_bits;               /* numero bits rappresentazione */
  int bit, ib;

  number = 8;
  n_bits = 4;

  printf("\n %d -> ", number);

  for (ib = 0; ib < n_bits; ++ib) {
    bit     = number % 2;             /* resto divisione (bit)  */
    printf("%d", bit);

    number /= 2;                      /* parte intera divisione */
  }
  printf(" \n\n");
  return 0;
}
```

Il programma inizia con il commento

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale unsigned.
```

```
 *
 * $yaC-Primer: unsigned2bin_0.c v 1.0 27.01.05 AC $
 */
```

che contiene alcune informazioni per il programmatore "umano". È buona norma di programmazione mettere all'inizio alcune informazioni sul programma per semplificare il mantenimento dei programmi.

Dopo il commento viene incluso il file di header `stdio.h`

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

perché si vuole utilizzare la funzione `printf()` per scrivere la rappresentazione binaria del numero sullo schermo.

Questo programma è molto semplice ed è costituito dalla sola funzione `main()`

```
int main(void)
{
....
  return 0;
}
```

La funzione `main()` deve restituire un valore di tipo `int` anche nel caso in cui il valore non venga utilizzato. Di conseguenza la funzione termina con l'istruzione

```
return 0;
```

che termina l'esecuzione della funzione e restituisce il valore 0. Se il valore restituito dalla funzione `main()` non viene utilizzato il valore restituito è irrilevante. Noi useremo la convenzione standard che associa al valore 0 la fine *corretta* dell'esecuzione della funzione.

Il corpo della funzione `main()` racchiuso dalle parentesi graffe "{}" inizia con la dichiarazione delle variabili che saranno utilizzate.

```
int  number;              /* numero decimale positivo     */
int  n_bits;              /* numero bits rappresentazione */
int bit, ib;
```

Lo Standard C89 richiede che tutte le dichiarazioni siano fatte all'inizio del corpo. È possibile includere alcuni commenti per specificare l'utilizzo delle variabili. In questo caso il commento indica che la variabile `number` contiene il valore del numero di cui si vuole la rappresentazione binaria mentre la variabile `n_bits` il numero di bits da utilizzare nella rappresentazione.

Le seguenti due istruzioni

```
number = 8;
n_bits = 4;
```

assegnano un valore alle variabili `number` e `n_bits`.

L'istruzione

```
printf("\n %d -> ", number);
```

scrive sul terminale il valore del numero del numero di cui si vuole la rappresentazione binaria in rappresentazione decimale. Osserviamo che mancando un *newline* alla fine della stringa l'output seguente continuerà sulla stessa riga senza andare a capo.

Il ciclo `for` che implementa l'algoritmo descritto per trovare la rappresentazione binaria di un numero espresso in rappresentazione decimale. L'istruzione

```
bit = number % 2;
```

assegna alla variabile `bit` il valore del resto della divisione del valore della variabile `number` per 2. La prima volta che questa istruzione viene eseguita la variabile `number` contiene il valore del numero, in questo caso il valore 8, per cui la variabile `bit` conterrà il valore del primo bit della rappresentazione binaria. Il bit della rappresentazione viene scritto sullo schermo utilizzando l'istruzione

```
printf("%d", bit);
```

La prossima istruzione

```
number /= 2;
```

assegna alla variabile `number` la parte intera della divisione del valore contenuto in `number` per 2: la variabile `number` è di tipo `int` per cui nella divisione per 2 la parte dopo il punto decimale, ossia il resto, viene eliminata. Se adesso si ripetono le prime due istruzioni del ciclo verrà quindi scritto il secondo bit della rappresentazione. È chiaro che l'uso della stessa variabile `number` sia per il numero originario che per la parte intera delle divisioni successive permette di scrivere facilmente il ciclo.

Infine l'istruzione

```
for (ib = 0; ib < n_bits; ++ib) {
   ...
}
```

ripete le tre istruzioni tante volte quanti sono i bits della rappresentazione voluti fornendo così la rappresentazione binaria cercata.

Chiaramente per poter cambiare il valore di `number` e/o `n_bits` bisogna modificare e ricompilare il programma. Questo è piuttosto scomodo per cui anticipando l'uso delle funzioni di Input, il programma può essere scritto in forma più versatile come

Programma: `unsigned2bin_1.c`

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale unsigned.
 *
 * Input       : numero intero positivo
 *               numero di bits per la rappresentazione
 *
 * $yaC-Primer: unsigned2bin_1.c v 1.2 27.01.05 AC $
 */
#include <stdio.h>
```

```
int main(void)
{
  int  number;               /* numero decimale positivo     */
  int  n_bits;               /* numero bits rappresentazione */
  int bit, ib;

  printf("\nNumero decimale positivo: ");
  scanf("%d", &number);

  printf("Quanti bits             : ");
  scanf("%d", &n_bits);

  printf("\n %d -> ", number);

  for (ib = 0; ib < n_bits; ++ib) {
    bit     = number % 2;             /* resto divisione (bit)  */
    printf("%d", bit);

    number /= 2;                      /* parte intera divisione */
  }
  printf(" \n\n");
  return 0;
}
```

La sola differenza con il programma precedente sono le quattro istruzioni

```
printf("\nNumero decimale positivo: ");
scanf("%d", &number);

printf("Quanti bits             : ");
scanf("%d", &n_bits);
```

che sostituiscono l'assegnazione del valore alle variabili `number` e `n_bits`. Parleremo più dettagliatamente delle funzioni di Input/Output tra breve, per il momento basti sapere che queste istruzioni chiedono di inserire da tastiera il numero di cui si vuole la rappresentazione binaria ed il numero di bits da utilizzare ed assegnano i valori letti rispettivamente alle variabili `number` e `n_bits`.

Questi due programmi non sono completamente corretti infatti sebbene l'algoritmo sia scritto correttamente entrambi scrivono i bits della rappresentazione in ordine inverso! Ad esempio il risultato del programma `unsigned2bin_1.c` è

```
Numero decimale positivo: 8
Quanti bits             : 4

 8 -> 0001
```

Il motivo è che la funzione `printf()` scrive da sinistra a destra partendo dalla posizione dell'ultimo carattere scritto. L'errore può essere corretto in molti modi, qui useremo la strategia di spostare la posizione di stampa a destra di tanti posti quanti sono i bits da scrivere per poi scriverli tornando indietro da destra a sinistra. Questo può essere ottenuto

facilmente utilizzando il *backspace* "\b" nella stringa stampata dalla funzione `printf()`, come illustrato nel programma seguente.

Programma: `unsigned2bin_2.c`

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale unsigned.
 *
 * Input       : numero intero positivo
 *               numero di bits per la rappresentazione
 *
 * $yaC-Primer: unsigned2bin_2.c v 1.1 24.01.04 AC $
 */
#include <stdio.h>

int main(void)
{
  int  number;               /* numero decimale positivo     */
  int  n_bits;               /* numero bits rappresentazione */
  int bit, ib;

  printf("\nNumero decimale positivo: ");
  scanf("%d", &number);

  printf("Quanti bits             : ");
  scanf("%d", &n_bits);

  printf("\n %d -> ", number);

  /* cursore a destra di n_bits posti */
  for (ib = 0; ib < n_bits; ++ib) printf(" ");

  for (ib = 0; ib < n_bits; ++ib) {
    bit     = number % 2;             /* resto divisione        */
    printf("%d\b\b", bit);            /* scrittura dx -> sx     */

    number /= 2;                      /* parte intera divisione */
  }
  printf(" \n\n");
  return 0;
}
```

La stampa da destra a sinitra è ottenuta con l'istruzione

```
    printf("%d\b\b", bit);            /* scrittura dx -> sx     */
```

Osserviamo che servono due *backspaces* per muoversi alla posizione a sinistra dell'ultimo carattere stampato.

Adesso i bits sono scritti correttamente, ma è rimasto ancora un problema. Il programma fornisce infatti la risposta corretta solo se il numero di bits è sufficiente per la rappresentazione

del numero. Nel caso contrario i bits in più non vengono considerati

```
Numero decimale positivo: 10
Quanti bits             : 3

 10 ->  010
```

senza che nessun avvertimento sia dato rendendo così il programma inaffidabile. La seguente è una versione migliorata del programma in cui viene effettuato il controllo sul numero massimo rappresentabile con il numero di bits richiesti.

Programma: `unsigned2bin_3.c`

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale unsigned.
 *
 * Input       : numero intero positivo
 *               numero di bits per la rappresentazione
 *
 * $yaC-Primer: unsigned2bin_3.c v 2.1 03.03.05 AC $
 */
#include <stdio.h>

int main(void)
{
  int      number;         /* numero decimale positivo       */
  int      n_bits;         /* numero bits rappresentazione   */
  int  number_max;         /* numero massimo rappresentabile */
  int     bit, ib;

  printf("\nNumero decimale positivo: ");
  scanf("%d", &number);

  printf("Quanti bits             : ");
  scanf("%d", &n_bits);

  /*
   * Controllo numero bits
   */

  /* numero massimo = 2^n_bits - 1 */
  number_max = 1;
  for (ib = 0; ib < n_bits; ++ib) number_max *= 2;
  number_max -= 1;

  if ( number > number_max) {
    printf("Non abbastanza bits !!!\n\n");
    return 1;
  }

  printf("\n %d -> ", number);
```

```
  /* cursore a destra di n_bits posti */
  for (ib = 0; ib < n_bits; ++ib) printf(" ");

  for (ib = 0; ib < n_bits; ++ib) {
    bit    = number % 2;          /* resto divisione          */
    printf("%d\b\b", bit);        /* scrittura dx -> sx       */

    number /= 2;                  /* parte intera divisione */
  }
  printf(" \n\n");
  return 0;
}
```

Diversamente da altri linguaggi di programmazione il linguaggio C non ha un'istruzione per calcolare la potenza di un numero, di conseguenza nel programma il valore di $2^{\texttt{n\_bits}} - 1$ viene calcolato direttamente moltiplicando 2 per `n_bits` volte e sottraendo 1 al risultato:

```
/* numero massimo = 2^n_bits - 1 */
number_max = 1;
for (ib = 0; ib < n_bits; ++ib) number_max *= 2;
number_max -= 1;
```

In alternativa si può utilizzare la funzione delle librerie matematiche standard

```
double pow(double x, double y);
```

il cui valore, di tipo `double`, è dato dal valore di `x` elevato alla potenza `y` e sostituire alle istruzioni precedenti l'istruzione

```
/* numero massimo = 2^n_bits - 1 */
number_max = pow(2, n_bits) - 1;
```

Siccome le variabili (e le costanti) sono di tipo `int`, e quindi di tipo diverso da quello "desiderato" dalla funzione `pow()`, il linguaggio C provvede ad effettuare le dovute conversioni dei valori nei tipi richiesti. Diremo di più sulle conversioni e *casts* tra breve.

Come nel caso della funzione `printf()`, anche per utilizzare la funzione `pow()` bisogna prima fornire al compilatore le informazioni utili sulla funzione. Queste sono contenute nel file di `header` di sistema `math.h` che quindi va incluso con l'istruzione "`#include`" del preprocessore:

```
#include <math.h>
```

Inoltre siccome la libreria matematica non è inclusa automaticamente dal compilatore è necessario richiedere al linker di farlo. Questo si ottiene con il *flag* "`-l`" del compilatore. Per cui se si utilizza questa soluzione il programma va compilato con

```
$ cc unsigned2bin_3.c -lm
```

dove il carattere "`m`" indica la libreria matematica.

Se adesso inavvertitamente (o volutamente) richiedessimo un numero insufficiente di bits per rappresentare il numero l'errore verrebbe segnalato

```
Numero decimale positivo: 10
Quanti bits             : 3
Non abbastanza bits !!!
```

e l'esecuzione del programma terminerebbe ritornando contemporaneamente al sistema il valore 1 interpretabile all'occorenza come *codice di errore*.

Osserviamo che il problema della scrittura del programma è stato suddiviso in passi successivi: la realizzazione dell'algoritmo, la lettura dei dati di input, la scrittura dei risultati ed infine il controllo su eventuali errori dovuti all'uso improprio del programma. Questo modo di procedere dividendo il problema principale in problemi più semplici che vengono risolti separatamente è un buon modo di procedere e permette di affrontare facilmente problemi molto complessi. Inoltre, siccome ogni singolo problema può essere controllato separatamente, è molto più facile scoprire gli eventuali errori. Una procedimento di questo genere è alla base di una buona programmazione strutturata e dovrebbe essere sempre utilizzato nello sviluppo dei programmi.

2.12. Esempio: Rappresentazione binaria di un numero intero decimale di tipo signed (Rev. 2.1)

Il programma dell'esempio precedente può essere facilmente modificato per fornire la rappresentazione binaria di un numero intero decimale di tipo `signed` in una delle varie codifiche utilizzate. In questo esempio verranno considerate le codifiche in true magnitude, in complemento a uno ed in complemento a due.

Codifica in true magnitude

La codifica in true magnitude con n bits utilizza il bit più a sinistra (*most significative bit*) come bit del segno: bit 0 numero positivo, bit 1 valore negativo. I restanti $n-1$ bits sono utilizzati per la rappresentazione binaria del modulo o valore assoluto del numero.

Il seguente programma fornisce la codifica in true magnitude del numero decimale letto da tastiera con un numero di bits specificato.

Programma: `signed2bin_tm.c`

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale signed
 *               codifica in true magnitude.
 *
 * Input       : numero intero decimale
 *               numero di bits per la rappresentazione
 *
 * $yaC-Primer: signed2bin_tm.c v 2.0 28.01.05 AC $
 */
#include <stdio.h>
```

```c
#include <math.h>

int main(void)
{
  int  number;
  int  n_bits;         /* numero bits rappresentazione   */
  int mod_max;         /* massimo modulo rappresentabile */
  int sig_bit;         /* bit del segno                  */
  int bit, ib;

  printf("\nNumero decimale: ");
  scanf("%d", &number);

  printf("Quanti bits    : ");
  scanf("%d", &n_bits);

  --n_bits;                       /* modulo un bit in meno */

  /*
   * bit del segno e modulo del numero
   */
  sig_bit = 0;                          /* bit del segno     */
  if (number < 0) {
    sig_bit =  1;
    number *= -1;                       /* modulo del numero */
  }

  /*
   * Controllo numero bits
   */

  /* modulo massimo = 2^n_bits - 1 */
  mod_max = pow(2, n_bits) - 1;
  if (number > mod_max) {
    printf("Non abbastanza bits !!!\n\n");
    return 1;
  }

  printf("\n %d -> ", number);

  /* cursore a destra di (n_bits + 2) posti */
  for (ib = -2; ib < n_bits; ++ib) printf(" ");

  /* modulo */
  for (ib = 0; ib < n_bits; ++ib) {
    bit     = number % 2;             /* resto divisione        */
    printf("%d\b\b", bit);            /* scrittura dx -> sx     */

    number /= 2;                      /* parte intera divisione */
  }

  /* bit del segno */
```

```
    printf("\b%d \n\n", sig_bit);
    return 0;
}
```

Note sul programma: `signed2bin_tm.c`

- `--n_bits;`

Un bit viene usato per il segno per cui si deve diminuire di 1 il numero di bits a disposizione per rappresentare il modulo del numero.

- `for (ib = -2; ib < n_bits; ++ib) printf("␣");`

Per stampare i bits nell'ordine corretto da destra a sinistra si sposta il "cursore" a destra di `n_bits+2` posizioni: `n_bits` posizioni per il modulo, una posizione per lasciare uno spazio tra il bit del segno ed i bits del modulo in modo da semplificare la lettura ed una posizione per il bit del segno. Per evitare di dover calcolare ogni volta il valore di `n_bits + 2` si fa partire il contatore del ciclo da -2 invece che da 0. Il risultato è lo stesso ma il numero di operazioni, e quindi il tempo di esecuzione, è minore. In questo caso particolare la differenza è impercettibile, ma nel caso di cicli molto lunghi la differenza può divenire apprezzabile.

- ```
  sig_bit = 0;
  if (number < 0) {
    sig_bit =  1;
    number *= -1;
  }
  ```

Queste istruzioni determinano il valore del bit del segno e del modulo del numero. Se il numero è non negativo il valore del bit del segno `sig_bit` è `0` altrimenti è `1`. In questo secondo caso si calcola anche il modulo del numero semplicemente moltiplicando il valore della variabile `number` per `-1`.

Alternativamente il modulo del numero può essere calcolato con l'istruzione

```
number = abs(number);
```

che utilizza la funzione

```
int abs(int j);
```

delle librerie standard che restituisce il valore assoluto del suo argomento. Le informazioni per utilizzare questa funzione si trovano nel file di header `stdlib.h` che va quindi incluso.

In entrambi i casi la variabile `number` conterrà il valore assoluto del numero la cui rappresentazione binaria è ottenuta utilizzando l'algoritmo discusso nel caso di un intero decimale di tipo `unsigned`.

- `printf("\b%d␣\n\n", sig_bit);`
```

Per ultimo si stampa il valore del bit del segno lasciando uno spazio bianco per separarlo dai bits del modulo.

Il programma usa la funzione `pow()` per cui va compilato specificando "`-lm`"

```
$ cc signed2bin_tm.c -lm
```

per informare il linker di includere le librerie matematiche.

Quando il programma viene eseguito si ha

```
Numero decimale: 5
Quanti bits    : 8

 5 ->  0 0000101
```

per i numeri positivi, e

```
Numero decimale: -5
Quanti bits    : 8

-5 ->  1 0000101
```

per quelli negativi.

Codifica in complemento a uno

La codifica in complemento a uno come la codifica in true magnitude utilizza un bit del segno ed $n-1$ bits per il modulo ma differisce da quest'ultima per il fatto che i valori negativi sono codificati negando tutti i bits della rappresentazione binaria del modulo. Il programma che fornisce la codifica in complemento a uno è quindi una semplice modifica del programma precedente.

Programma: `signed2bin_c1.c`

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale signed
 *               codifica in complemento a uno.
 *
 * Input       : numero intero decimale
 *               numero di bits per la rappresentazione
 *
 * $yaC-Primer: signed2bin_c1.c v 1.0 28.01.05 AC $
 */
#include <stdio.h>
#include <math.h>

int main(void)
{
  int  number;
  int  n_bits;         /* numero bits rappresentazione   */
```

```
int mod_max;          /* massimo modulo rappresentabile */
int sig_bit;          /* bit del segno                  */
int bit, ib;

printf("\nNumero decimale: ");
scanf("%d", &number);

printf("Quanti bits    : ");
scanf("%d", &n_bits);

--n_bits;                       /* modulo un bit in meno */

/*
 * bit del segno e modulo del numero
 */
sig_bit = 0;                          /* bit del segno     */
if (number < 0) {
  sig_bit =  1;
  number *= -1;                       /* modulo del numero */
}

/*
 * Controllo numero bits
 */

/* modulo massimo = 2^n_bits - 1 */
mod_max = pow(2, n_bits) - 1;
if (number > mod_max) {
  printf("Non abbastanza bits !!!\n\n");
  return 1;
}

printf("\n %d -> ", number);

/* cursore a destra di (n_bits + 2) posti */
for (ib = -2; ib < n_bits; ++ib) printf(" ");

/* modulo */
for (ib = 0; ib < n_bits; ++ib) {
  bit     = number % 2;               /* resto divisione        */

  /* Se negativo nega bit */
  if (sig_bit == 1)
    bit = (bit == 0) ? 1 : 0;

  printf("%d\b\b", bit);              /* scrittura dx -> sx     */

  number /= 2;                        /* parte intera divisione */
}

/* bit del segno */
printf("\b%d \n\n", sig_bit);
```

```
    return 0;
}
```

Note sul programma: `signed2bin_c1.c`

- `if (sig_bit == 1) bit = (bit == 0) ? 1 : 0;`

  Se il numero è negativo il bit va negato, per cui se è `0` diventa `1` e viceversa. La negazione è ottenuta utilizzando l'espressione condizionata

  ```
  (bit == 0) ? 1 : 0
  ```

  il cui valore è `1` o `0` a seconda che `bit` sia `0` o `1`. Alternativamente si poteva utilizzare l'istruzione

  ```
  bit = 1 - bit;
  ```

  che, come è facile convincesi, ha lo stesso effetto.

  Osserviamo che siccome nel linguaggio C un qualsiasi valore non nullo è considerato *true* si sarebbe potuto utilizzare la scrittura compatta

  ```
  if (sig_bit) bit = (!bit) ? 1 : 0;
  ```

  Il risultato è lo stesso, la leggibilità no.

Il modo più semplice per controllare la correttezza di un programma è quella di utilizzarlo, se possibile, con dei casi di cui si conosce il risultato. Ad esempio in questo caso si può calcolare la rappresentazione con 8 bits dei numeri 5 e -5. Il programma fornisce per questi due casi

```
Numero decimale: 5
Quanti bits    : 8

 5 ->  0 0000101
```

per il valore positivo, e

```
Numero decimale: -5
Quanti bits    : 8

-5 ->  1 1111010
```

per il valore negativo. Un rapido confronto tra i due risultati, e con la codifica in true magnitude discussa prima, mostra che il programma fornisce il risultato corretto.

Codifica in "complemento a due"

Anche la rappresentazione in complemento a due utilizza il bit più a sinistra per il segno ed i restanti $n-1$ bits per il modulo ma la sua codifica è differente per i valori positivi e negativi. Per i primi il modulo è codificato con l'usuale rappresentazione binaria. Per i numeri

negativi invece il modulo viene codificato negando ciascun bit della rappresentazione binaria del modulo ed aggiungendo 1 al risultato. Non è difficile verificare che se si usano n bits allora la codifica in complemento a due di $(-k)$ $(k > 0)$ è data dalla rappresentazione binaria con n bits del numero positivo $2^n + (-k)$ che può essere determinata utilizzando l'algoritmo discusso nell'esempio precedente. Il programma per la codifica in complemento a due è quindi una semplice modifica dei programmi precedenti.

Programma: `signed2bin_c2.c`

```
/*
 * Descrizione : rappresentazione binaria di un numero intero
 *               decimale signed
 *               codifica in complemento a due.
 *
 * Input       : numero intero
 *               numero di bits per la rappresentazione
 *
 * $yaC-Primer: signed2bin_c2.c v 1.2 28.01.05 AC $
 */
#include <stdio.h>
#include <math.h>

int main(void)
{
  int  number;
  int  n_bits;         /* numero bits rappresentazione   */
  int mod_max;         /* massimo modulo rappresentabile */
  int bit, ib;

  printf("\nNumero decimale: ");
  scanf("%d", &number);

  printf("Quanti bits    : ");
  scanf("%d", &n_bits);

  /*
   * Controllo numero bits
   */

  /* range = -2^(n_bits - 1), ..., 2^(n_bits - 1) - 1 */
  mod_max = pow(2, n_bits - 1);
  if ( (number < -mod_max) || (number > mod_max - 1) ) {
    printf("Non abbastanza bits !!!\n\n");
    return 1;
  }

  printf("\n %d -> ", number);

  /* cursore a destra di (n_bits + 1) posti */
  for (ib = -1; ib < n_bits; ++ib) printf(" ");
```

```
  /* Numero negativo */
  if (number < 0)  number += 2 * mod_max;

  /* primi (n_bits - 1) bits */
  for (ib = 1; ib < n_bits; ++ib) {
    bit     = number % 2;             /* resto divisione        */
    printf("%d\b\b", bit);            /* scrittura dx -> sx     */

    number /= 2;                      /* parte intera divisione */
  }

  /* bit del segno */
  printf("\b%d \n\n", number % 2);
  return 0;
}
```

Note sul programma: `signed2bin_c2.c`

- ```
  mod_max = pow(2, n_bits - 1);
  if ( (number < -mod_max) || (number > mod_max - 1) ) {
    printf("Non abbastanza bits !!!\n\n");
    return 1;
  }
  ```

  Con n bits e la codifica in complemento a due si possono rappresentare i valori da -2^{n-1} fino a $2^{n-1}-1$ inclusi per cui il controllo deve essere fatto separatamente per i valori positivi e quelli negativi. I due controlli possono essere uniti insieme con l'operatore logico OR "`||`".

- `if (number < 0)  number += 2 * mod_max ;`

  Se il numero è negativo la sua codifica in complemento a due è uguale alla rappresentazione binaria del numero positivo $2^{\texttt{n\_bits}} + \texttt{number}$ (`number` è negativo!). Il valore di $2^{\texttt{n\_bits}}$ è ottenuto moltiplicando per 2 il valore di $\texttt{mod\_max} = 2^{\texttt{n\_bits}-1}$.

- `printf("\b%d␣\n\n", number % 2);`

  Come nel programma precedente il bit del segno e quelli del modulo vengono separati da uno spazio.

Il controllo del risultato si effettua facilmente verificando che la somma bit a bit delle rappresentazioni di due numeri k e $-k$ con n bits faccia 0 (con n bits!). Se ad esempio consideriamo i numeri 5 e -5 ed 8 bits il programma fornisce le rappresentazioni:

```
Numero decimale: 5
Quanti bits    : 8

 5 ->  0 0000101
```

e
```

```
Numero decimale: -5
Quanti bits    : 8

 -5 ->  1 1111011
```

da cui si vede facilmente che il risultato della loro somma bit a bit è una sequenza di 8 bit 0 che altro non è che la rappresentazione con 8 bits del valore 0.

2.13. Conversioni (Rev. 2.1)

Il linguaggio C permette di definire oggetti di tipo differente che in genere assumono un insieme di valori diversi. Nasce quindi naturalmente il problema di dover convertire i valori di un tipo in valori di un altro tipo. La conversione è ad esempio necessaria quando gli operandi di un operatore binario sono di tipo diverso. In questo caso infatti prima di effettuare l'operazione bisogna trasformare i valori dei due operandi in valori di uno stesso tipo e poi effettuare l'operazione.

Il problema della conversione dei valori da un tipo all'altro è meno immediato di quanto possa sembrare a prima vista perché coinvolge sia l'insieme dei valori che i due tipi possono assumere che la loro rappresentazione. Ad esempio il tipo `int` ed il tipo `double` non differiscono solo per l'insieme dei valori che possono prendere ma anche nella rappresentazione utilizzata.

In questo Primer non ci addentreremo troppo nel problema della conversione dei valori da un tipo all'altro ma ci limiteremo a dare delle indicazioni di carattere generale sufficienti a livello di programmazione. In particolare qui tratteremo solo le conversioni tra tipi interi e floating-point.

Prima di passare alla discussione dei vari tipi di conversione è bene puntualizzare che tutte le conversioni sono effettuate solo sui valori delle espressioni e non sulle espressioni stesse che mantengono quindi il proprio tipo. Questo vuol dire, ad esempio, che il valore di una variabile di tipo `int` può essere convertito in tipo `float` ma il tipo della variabile rimarrà in ogni caso `int`.

2.13.1. Conversione a tipo Intero

Da tipo intero La regola generale per la conversione da tipo intero a tipo intero è che il valore convertito sia uguale al valore originale. Ad esempio se il valore 10 di un tipo intero senza segno viene convertito in un tipo intero con segno il valore dopo la conversione deve essere ancora 10.

Se non è possibile rappresentare il valore nel nuovo tipo il risultato dipende dal tipo. Se il nuovo tipo è con segno si ha un *overflow* ed il risultato è indefinito. Se invece il nuovo tipo è di tipo senza segno il valore del risultato è il valore originale $\bmod 2^n$, ossia il resto della divisione del valore per 2^n, dove n è il numero di bits utilizzati per rappresentare i valori del nuovo tipo. Ad esempio se il valore originale è 5 ed il nuovo tipo senza segno utilizza $n = 2$ bits il valore dopo la conversione è $5 \bmod 2^2 = 1$.

Da tipo floating-point La regola generale per la conversione da tipo floating-point a tipo intero è che il valore convertito sia il più possibile uguale al valore originale. Se il tipo floating-point ha una parte frazionaria questa viene semplicemente trascurata. La conversione è quindi equivalente al troncamento del valore floating-point alla parte intera.

Se il valore floating-point non può essere rappresentato come tipo intero, ad esempio troppo grande o troppo piccolo, il risultato della conversione è indeterminato e può dipendere dal sistema.

Tipo Booleano (C99) Il C99 ha introdotto il tipo intero `_Bool`. Le conversioni da e a tipo `_Bool` sono leggermente differenti da quelle con gli altri tipi interi. Infatti nella conversione a tipo `_Bool` il valore convertito è 0 se il valore originale è nullo altrimenti è sempre 1. Nelle conversioni da tipo `_Bool` il valore è 0 o 1 convertito nel nuovo tipo.

2.13.2. Conversione a tipo Floating-Point

Da tipo intero Nella conversione da tipo intero a tipo floating-point se il valore del tipo intero può essere rappresentato esattamente dal tipo floating-point il risultato è semplicemente il valore intero nella rappresentazione del tipo floating-point.

Se invece il valore del tipo intero non può essere rappresentato esattamente, ma il valore è nell'intervallo dei valori rappresentabili dal tipo floating-point, il risultato è il valore floating-point più vicino. Se invece il valore è al di fuori dell'intervallo dei valori rappresentabili dal tipo floating-point il risultato è indefinito.

Da tipo floating-point Nella conversione da tipo `float` a tipo `double` o da tipo `double` a tipo `long double` il risultato è il valore originale nella nuova rappresentazione.

Nella conversione da tipo `double` a tipo `float` o da tipo `long double` a tipo `double` il risultato è il valore floating-point nella nuova rappresentazione che più si avvicina al valore originale. Nella conversione il valore può essere arrotondato per eccesso o per difetto a seconda del sistema.

Se il valore originale è al di fuori dell'intervallo dei valori rappresentabili dal nuovo tipo floating-point il risultato è indefinito. In questi casi si ha un *overflow* o *underflow* a seconda che il valore sia troppo grande o troppo piccolo.

Tipi complessi (C99) La conversione tra tipi reali, sia interi che floating-point, e tipi complessi o tra tipi complessi segue regole piuttosto semplici.

Nella conversione tra due tipi complessi la parte reale e la parte immaginaria sono convertite separatamente secondo le regole della conversione dei valori floating-point reali.

Nella conversione di un tipo reale ad un tipo complesso la parte immaginaria del risultato viene posto a zero mentre il valore reale viene convertito secondo le regole di conversione dei valori floating-point reali ed assegnato alla parte reale del risultato.

Nella conversione inversa di un tipo complesso in un tipo reale la parte reale viene convertita

secondo le regole di conversione dei tipi reali, mentre la parte immaginaria viene semplicemente ignorata.

I tipi `_Imaginary` sono tipi complessi con la parte reale nulla, di conseguenza nella conversione di un tipo `_Imaginary` in un tipo complesso la parte reale del risultato è sempre nulla mentre la parte immaginaria segue le regole di conversione dei tipi floating-point reali. Viceversa la conversione di un tipo complesso in un tipo `_Imaginary` semplicemente trascura la parte reale. Infine la conversione di un tipo `_Imaginary` in un qualsiasi tipo reale è sempre zero.

2.13.3. Conversioni implicite

Il linguaggio C esegue a seconda delle circostanze tutta una serie di conversioni automatiche tra tipi diversi chiamate genericamente *conversioni implicite*.

Conversioni Unarie. Per ridurre il numero di tipi numerici che devono essere manipolati dagli operatori, e quindi di conseguenza ridurre il numero degli operatori, il linguaggio C effettua in alcuni casi una conversione implicita dei valori degli operandi degli operatori prima di effettuare una qualsiasi operazione. Ad esempio se `i` e `j` sono di tipo `short` il loro valore viene convertito a tipo `int` prima di effettuare qualsiasi operazione aritmetica, cosicché il valore valore di `i+j` non è di tipo `short` ma di tipo `int`. Queste conversioni implicite sono chiamate *conversioni unarie* perché coinvolgono un solo operando.

Conversioni Unarie Aritmetiche

Operando		**Conversione**
`char` o `short` o `signed char`	⇒	`int`
`unsigned char` o `unsigned short`	⇒	`int` o `unsigned int` se il tipo `int` non può rappresentare i valori originali
`_Bool` (C99)	⇒	`int`

Nel Traditional C il tipo `float` veniva convertito automaticamente in ogni espressione numerica a tipo `double` per ridurre il numero di funzioni floating-point delle librerie standard. Nello Standard ISO C non vi è più questa richiesta per permettere di utilizzare operazioni in singola precisione, meno precise ma generalmente più veloci delle corrispondenti operazioni in doppia precisione. Lo Standard ISO C lascia comunque libertà ai compilatori di continuare ad effettuare la conversione dei valori di tipo `float` a tipo `double` nelle espressioni aritmetiche. Su alcuni compilatori la conversione automatica unaria da `float` a `double` può essere attivata o disattivata mediante opportune *flags*.

Conversioni Binarie. Oltre alle conversioni unarie il linguaggio C effettua anche *conversioni binarie* che coinvolgono tutti gli operandi di un operatore. La necessità di queste conversioni implicite nasce dal fatto che gli operatori binari richiedono che gli operandi siano dello stesso

tipo per cui ogni qual volta nelle assegnazioni o nelle espressioni vi sono termini di tipo diverso è necessario convertirne i valori ad un tipo comune prima di effettuare l'operazione. Le conversioni implicite sono differenti a seconda che si tratti di una assegnazione o di un'operazione aritmetica.

Nelle assegnazioni la conversione implicita viene sempre effettuata per convertire il valore da assegnare nel tipo della variabile a cui va assegnato. Questo tipo di conversione viene effettuato anche sui valori degli argomenti delle chiamate a funzione prima di assegnarli ai parametri della funzione, come ad esempio nella chiamata a funzione nell'istruzione

```
number_max = pow(2,n_bits) - 1;
```

utilizzata nel programma per la rappresentazione dei numeri interi decimali con segno. In questo caso infatti siccome la funzione `pow()` è dichiarata come

```
double pow(double x, double y);
```

i valori della costante `2` di tipo intero e della variabile `n_bits` di tipo `int` sono convertiti in valori di tipo `double` prima di assegnarli ai parametri `x` e `y` di tipo `double` della funzione.

Nella valutazione delle espressioni aritmetiche invece le conversioni implicite sono effettuate in modo da evitare di perdere informazioni per cui le conversioni sono effettuate sempre verso i tipi che possono rappresentare l'insieme più grande di valori. Di conseguenza nel caso in cui entrambi gli operandi siano interi, dopo la conversione unaria dei valori di tipo `char` e `short` sia con segno che senza segno, vengono applicate in ordine di precedenza le seguenti conversioni:

Conversioni Binarie Aritmetiche (C89)

Un operando	**Altro operando**		**Entrambi convertiti a**
`unsigned long`	tipo intero	⇒	`unsigned long`
`long`	`unsigned`	⇒	`long` o `unsigned long` se il tipo `long` non può rappresentare il valore `unsigned`
`long`	`int`	⇒	`long`
`unsigned`	`int` o `unsigned`	⇒	`unsigned`

Nello Standard C99 vi sono i tipi interi aggiuntivi `long long int`, `unsigned long long int` e `_Bool`. Questi non sono stati inclusi nella precedente tavola per semplicità. Se si volessero includere la regola è semplice. In tutte le conversioni con due operandi di tipo intero, una volta eseguite le conversioni unarie, i valori dei due operandi sono convertiti nel tipo dell'operando che può rappresentare i valori di entrambi i tipi. Se questo non è possibile entrambi vengono convertiti nel tipo intero `unsigned` immediatamente più grande che può rappresentare entrambi i valori. Ad esempio nella tavola precedente se il tipo `long` non può rappresentare il valore di tipo `unsigned` entrambi vengono convertiti nel tipo `unsigned long`.

Nel caso in cui uno dei due operandi sia di tipo floating-point tutte le operazioni vengono fatte in floating-point e l'altro operando viene convertito in ordine secondo le seguenti regole:

Conversioni Binarie Aritmetiche

Un operando	Altro operando		Entrambi convertiti a
`long double`	tipo reale[a]	⇒	`long double`
`double`	tipo reale[a]	⇒	`double`
`float`	tipo reale[a]	⇒	`float`

[a] Sia di tipo intero che floating-point.

Le conversioni binarie con i tipi complessi introdotti dallo Standard C99 seguono la regola generale delle conversioni binarie con valori floating-point reali. Quando entrambi gli operandi sono di tipo complesso il valore del tipo più corto viene convertito nel tipo più lungo. Se uno dei due operandi è di tipo complesso e l'altro di tipo reale il valore dell'operando reale e della parte reale ed immaginaria dell'operando complesso sono convertiti ad uno stesso tipo floating-point secondo le regole di conversione dei valori floating-point reali. L'operazione viene poi eseguita come se l'operando reale fosse stato convertito al tipo complesso di uguale precisione floating-point del valore reale.

Analizzando in conclusione l'istruzione

```
number_max = pow(2,n_bits) - 1;
```

si nota che si hanno quattro conversione implicite. Infatti per prima cosa il valore della costante intera `2` e quello della variabile intera `n_bits` sono convertiti nel tipo `double`, poi il valore della costante intera `1` viene convertito nel tipo `double` per poter effettuare la somma con il valore di tipo `double` della funzione `pow()`. Infine il valore della somma viene convertito nel tipo `int` ed assegnato alla variabile `max_number` di tipo `int`.

2.13.4. Divisione tra tipi interi e tra tipi floating-point

L'operatore di divisione "`/`" richiede qualche cautela in più poiché vi è una grande differenza tra la divisione tra tipi interi e tra tipi floating-point. Nel caso di divisione tra tipi interi infatti il valore dell'operatore è ancora un tipo intero per cui eventuali cifre dopo il punto decimale non vengono prese in considerazione. Queste vengono semplicemente omesse senza nessun tipo di arrotondamento. In altre parole il valore della divisione tra due tipi interi è dato dalla parte intera della divisione.

Se almeno uno dei due operandi (divisore o dividendo) è di tipo floating-point il valore dell'altro operando viene convertito automaticamente, se necessario, in valore tipo floating-point e la divisione viene effettuata in floating-point. Il valore dell'operatore "`/`" è in questo caso di tipo floating-point. Ad esempio

Espressione		Valore	Tipo
`19 / 10`	⇒	`1`	intero
`19 / 10.0`	⇒	`1.9`	floating-point
`1 / 2`	⇒	`0`	intero

Espressione		Valore	Tipo
`1. / 2`	⇒	0.5	floating-point
`1. / 2.`	⇒	0.5	floating-point

Il seguente frammento di programma fornisce ulteriori esempi di conversioni automatiche nelle espressioni che coinvolgono l'operatore di divisione "`/`" e di assegnazione "`=`".

```
int   int_v;
float float_v;

float_v = 1.0 / 2.0;          /* assegna il valore 0.5 a float_v */
int_v   = 1 / 3;              /* assegna il valore 0   a int_v   */
float_v = (1 / 2) + (1 / 2);  /* assegna il valore 0.0 a float_v */
float_v = (1 / 2) + (1./ 2);  /* assegna il valore 0.5 a float_v */
int_v   = (1 / 2) + (1./ 2);  /* assegna il valore 0   a int_v   */
int_v   = 3.0 / 2;            /* assegna il valore 1   a int_v   */
float_v = 3.0 / 2;            /* assegna il valore 1.5 a float_v */
int_v   = float_v;            /* assegna il valore 1   a int_v   */
```

La differenza del valore dell'operatore di divisione "`/`" per divisioni tra tipi interi e tra tipi floating-point è spesso causa di errori, a volte di difficile individuazione.

2.13.5. Conversioni esplicite

Il linguaggio C permette di effettuare conversioni esplicite, chiamate *cast*, mediante l'operatore di cast "`(type)`" nella forma

```
(type) expression
```

dove `type` è il tipo `T` in cui si vuole convertire il valore dell'espressione `expression`.

L'operatore di cast "`(type)`" è un operatore unario con un'associatività da destra e livello di precedenza superiore agli operatori matematici ma inferiore agli operatori unari di segno. Il valore dell'operatore di cast "`(type)`" è il valore del suo operando `expression` convertito a tipo `type`. L'operazione di conversione viene effettuata solo sul valore dell'espressione e non sull'espressione stessa, per cui viene prima valutato il valore di `expression` e poi questo, convertito nel tipo `type`, diviene il valore dell'operatore. Ad esempio se `i` è una variabile di tipo `int` allora il valore di "`(double) i`" è lo stesso di `i` ma di tipo `double`. La variabile `i` non viene modificata né nel valore né nel tipo. Di conseguenza un'istruzione del tipo

```
(double) i = 1.0;
```

è illegale. Per contro l'istruzione

```
(double) (i = 1)
```

è perfettamente legale in quanto prima viene assegnato il valore `1` alla variabile intera `1` e poi il valore dell'operatore di assegnazione, ossia `1`, viene trasformato in tipo `double`.

Operazioni di casting sono particolarmente utili in congiunzione con l'operatore di divisione "`/`", come illustra il seguente frammento di programma

```
int good = 5, bad = 3;
double  rate;

ratio = good / bad;                    /* ratio = 1.0      Errato!! */
ratio = (double) good / (double) bad; /* ratio = 1.666667 Corretto */
```

L'operatore di cast ha la precedenza sugli operatori matematici di conseguenza la conversione dei valori viene effettuato prima delle operazioni matematiche, come illustrato dal seguente programma.

Programma: `casting.c`

```
#include <stdio.h>

int main(void)
{
  double var_d;

  var_d = 3.0 / 2.0;
  printf("        3.0 / 2.0 => %f\n", var_d);

  var_d = (int) 3.0 / 2.0;
  printf("  (int) 3.0 / 2.0 => %f\n", var_d);

  var_d = ((int) 3.0) / 2.0;
  printf("((int) 3.0) / 2.0 => %f\n", var_d);

  var_d = (int) (3.0 / 2.0);
  printf("(int) (3.0 / 2.0) => %f\n", var_d);

  var_d = (int) 3.0 / 2;
  printf("    (int) 3.0 / 2 => %f\n", var_d);

  return 0;
}
```

Quando il programma viene eseguito sul terminale si ha il seguente output

```
        3.0 / 2.0 => 1.500000
  (int) 3.0 / 2.0 => 1.500000
((int) 3.0) / 2.0 => 1.500000
(int) (3.0 / 2.0) => 1.000000
    (int) 3.0 / 2 => 1.000000
```

Analizzando i valori risulta chiaro che, se non specificato diversamente, l'operazione di conversione viene effettuata prima della divisione. Particolarmente istruttiva è la differenza tra l'espressione

```
(int) 3.0 / 2.0
```

e l'espressione

```
(int) 3.0 / 2
```

In entrambi i casi il valore della costante `3.0` di tipo floating-point vene convertito dall'operatore di cast "`(int)`" in valore intero prima di effettuare la divisione. Tuttavia nella prima espressione il secondo operando dell'operatore di divisione è di tipo floating-point per cui il valore viene riconvertito in tipo floating-point con una conversione implicita prima di effettuare la divisione, il valore dell'espressione è quindi `1.5`. Nella seconda espressione invece il secondo operando dell'operatore di divisione è di tipo intero cosicché entrambi i valori degli operandi dell'operatore sono di tipo intero e la divisione viene effettuata tra tipi interi. Il valore dell'espressione è in questo caso `1`, la parte intera della divisione.

Conversioni esplicite possono essere fatte ogni qual volta in una espressione o assegnazione sia richiesto un valore di un tipo specificato per cui ad esempio al posto dell'istruzione

```
number_max = pow(2, n_bits) - 1;
```

si potrebbe utilizzare

```
number_max = (int) pow((double) 2, (double) n_bits) - 1;
```

per indicare esplicitamente le conversioni da effettuare, tre in questo caso e non quattro come nell'istruzione precedente. Sebbene in questo caso il cast non sia necessario in quanto le conversioni implicite effettuano le conversioni corrette, il suo uso rende l'istruzione più chiara per cui è facile trovare nei programmi operazioni di cast anche se non strettamente necessari. In ogni caso vale sempre la regola che se non si è sicuri della conversione implicita è meglio usare un cast.

2.14. Input/Output (Rev. 2.1.6)

Nel linguaggio C non vi sono istruzioni per le operazioni di Input / Output (I/O). Tutte le operazioni di I/O in C vengono gestite tramite funzioni delle librerie standard di sistema per cui vi sono molte possibilità diverse o alternative di effettuare le operazioni di I/O, qui noi considereremo solo le funzioni principali. Tutte le informazioni sulle funzioni di I/O si trovano nel file di header "`stdio.h`", che va quindi incluso ogni qualvolta si debbano effettuare operazioni di I/O.

Tutte le operazioni di I/O in C si basano sul concetto di *stream* (flusso o successione di dati) che può essere un file ma anche la tastiera o il terminale o qualsiasi altro *device* fisico sorgente o utilizzatore di dati come ad esempio schede di rete, modem etc. Lo stream introduce un nuovo tipo di dati `FILE` definito nel file di header "`stdio.h`". Il tipo `FILE` è un oggetto più complesso di quelli incontrati fino ad ora poiché deve contenere molte più informazioni come ad esempio la posizione corrente nello stream (*file position*), se vi sono stati errori di lettura o di scrittura o se si è raggiunta la fine dello stream (*end-of-file*), etc. Nel seguito i termini *file* e *stream* saranno utilizzati come sinonimi.

Per una maggiore efficienza gli streams possono utilizzare una o più zone di memoria (*buffers*) ad accesso veloce dove depositare blocchi di dati in modo che lo scambio di dati con il programma avvenga con i buffers e non direttamente con i devices (*buffered* I/O). Lo scambio

di dati tra i buffers e i devices avviene con processi più lenti ad esempio quando i dati contenuti nei buffers diventano obsoleti (buffers di lettura) oppure quando non vi è più spazio (buffers di scrittura) o con altri criteri a seconda del tipo di device. È possibile avere qualche controllo sui buffers utilizzando la funzione `setvbuf()`, anche se di solito ciò non è necessario in quanto gli streams sono generalmente realizzati in modo molto efficiente e quindi non ci si deve preoccupare di migliorarne l'efficienza.

Vi sono due tipi diversi di stream: stream di testo e stream binario. Gli streams di testo sono dati da una sequenza di linee, ciascuna delle quali composta da una sequenza di caratteri alfanumerici terminata dal carattere di *end-of-line*. Il carattere di end-of-line è considerato parte della linea. Lo Standard C richiede che uno stream di testo possa essere composto di linee di almeno 254 caratteri incluso il *newline*. Siccome usualmente si utilizzano caratteri ASCII gli stream di testo sono anche chiamati streams o files ASCII. Con lo Standard C89 Amendment 1 e C99 i caratteri possono anche essere di tipo *wide*.

Gli streams di testo possono in generale essere scambiati facilmente da un sistema all'altro, a patto che le linee utilizzino caratteri standard riconosciuti da entrambi i sistemi. Ad esempio se si usano caratteri ad 8-bits per rappresentare caratteri speciali come ad esempio 'è' o 'ü' lo stream di testo potrà essere scambiato solo se entrambi i sistemi usano la stessa codifica con 8-bits per i caratteri speciali. In caso contrario il risultato potrebbe essere piuttosto interessante.

Gli streams binari sono sequenze di dati di tipo `char`. Siccome ogni oggetto in C può essere rappresentato come una sequenza di valori di tipo `char` gli streams binari possono contenere qualsiasi tipo di dati, inclusi quelli contenuti negli streams di testo. In effetti non è necessario che gli streams binari e quelli di testo siano rappresentati differentemente, quello che cambia è infatti solo l'interpretazione dei dati contenuti, per cui su alcuni sistemi per motivi di efficienza non vi è differenza nelle loro rappresentazioni.

L'accesso ad uno stream binario è molto più rapido dell'accesso ad uno stream di testo perché i dati nei computers sono in forma binaria e quindi non è richiesta nessuna conversione di caratteri, inoltre a parità di informazioni uno stream binario di solito richiede meno memoria di uno stream di testo. Per contro uno stream binario non può essere visualizzato direttamente, basti pensare a quello che si otterrebbe cercando di visualizzare il contenuto del file `a.out`. Al contrario degli streams di testo, inoltre, gli streams binari difficilmente possono essere scambiati da un sistema ad un altro poiché la rappresentazione utilizzata per gli streams dipende in generale dal sistema. Anche la rappresentazione degli stream di testo dipende dal sistema, ma lo Standard C richiede che qualsiasi sia la rappresentazione utilizzata su un particolare hardware o software, gli streams di testo vengano comunque trasformati nel formato standard, linee di caratteri terminate dal *newline*, dalle routines di Input/Output dello stream.

A questo proposito ricordiamo il problema della rappresentazione del carattere di end-of-line. I sistemi UNIX usano per terminare una linea il carattere newline (`NL`) o line-feed "`\n`", codice ASCII `0x0A`. Apple, prima di passare ad un sistema basato su UNIX usava il return "`\r`" (`CR`), codice ASCII `0x0D`. MS-DOS/Windows usa due caratteri: il return (`CR`) ed il newline (`NL`). Il motivo è di origine storica e risale alla telescrivente meccanica "Teletype 33" che permetteva di inviare per telefono attraverso un modem una media di 10 caratteri al secondo. Il problema era che la telescrivente scriveva su un foglio di carta come una macchina

da scrivere e quando la testina raggiungeva la fine del foglio impiegava circa due decimi di secondo per posizionare la testina all'inizio della riga successiva. Due decimi di secondo era il tempo medio per inviare due caratteri per cui poteva accadere che il carattere successivo al carattere di fine linea arrivasse mentre la testina era ancora in moto e quindi veniva perso. Il problema fu risolto introducendo un end-of-line di due caratteri: il `CR` per posizionare la testina sul bordo sinistro del foglio e `NL` per far avanzare il foglio di una riga. Nei primi computers la memoria era una merce costosa ed utilizzare due caratteri era chiaramente uno spreco così uno dei due caratteri fu soppresso. Alcuni sistemi mantennero solo il `NL`, altri il `CR`, altri ancora non si preoccuparono del problema e continuarono ad usare due caratteri.

La morale di questa storia è che in MS-DOS/Windows vi è una differenza se un file viene aperto come stream di testo o binario. Nel primo caso il `RETURN` in più viene eliminato dalle routines di lettura, nel secondo caso invece il carattere è li e bisogna tenerne conto. Di conseguenza un programma che utilizzi stream binari dovrà trattarli diversamente a seconda che il sistema sia UNIX o MS-DOS/Windows.

Lo Standard C richiede che il numero massimo di streams che possono essere aperti simultaneamente sia contenuto in `FOPEN_MAX` e che non sia inferiore ad otto, inclusi i tre streams standard predefiniti `stdin`, `stdout` e `stderr`.

2.14.1. Streams standard: stdin, stdout, stderr

Ogni volta che l'esecuzione di un programma C inizia vengono aperti automaticamente tre streams predefiniti:

Identificatore		**stream**	**modo**
`stdin`	⇒	standard input	*read*
`stdout`	⇒	standard output	*write*
`stderr`	⇒	standard error	*write*

Sintatticamente le variabili `stdin`, `stdout` e `stderr` sono degli oggetti di tipo `FILE *`, ossia dei *puntatori* ad un oggetti di tipo `FILE`. Avremo molto da dire sui puntatori, per il momento ci basti sapere che il valore di queste variabili è l'indirizzo di memoria dove si trovano le informazioni relative allo stream corrispondente.

Quando i programmi sono eseguiti da terminale lo stream `stdin` è associato alla tastiera mentre gli streams `stdout` e `stderr` al terminale.

2.14.2. Dichiarazione, apertura e chiusura degli streams

Se si vuole utilizzare un stream differente da quelli standard questo va prima dichiarato, come tutti gli oggetti in C, ed poi aperto. La dichiarazione di uno stream è della forma

```
FILE *file_handle
```

dove il tipo `FILE` indica che si tratta di uno stream e `file_handle` è l'identificatore dello stream. Osserviamo che l'identificatore è preceduto dall'operatore unario "`*`", che in questo non indica una moltiplicazione. Sintatticamente la dichiarazione dichiara che `file_handle` è un *puntatore* ad un oggetto di tipo `FILE`, il che vuol dire che il valore della variabile `file_handle` è l'indirizzo di memoria dove sono contenute le informazioni relative allo stream. La dichiarazione definisce un identificatore `file_handle` che può essere utilizzato per accedere ad uno stream ma non gli associa un stream. Detto in altri termini la dichiarazione non inizializza la variabile `file_handle`.

Funzione fopen()

Per poter utilizzare uno stream specifico bisogna prima di tutto *aprirlo* scrivendo le informazioni relative allo stream da qualche parte nella memoria e poi assegnare l'indirizzo di memoria dove si trovano queste informazioni ad una variabile `file_handle` di tipo `FILE *` in modo da potervi accedere. Tutte queste operazioni sono fatte dalla funzione

```
#include <stdio.h>

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

per cui l'apertura di uno stream è della forma

```
file_handle = fopen(path, mode);
```

In questo primer utilizzeremo la convenzione standard di indicare insieme alle specifiche delle funzioni (*prototipo*) il file di header da includere per poterle utilizzare, in questo caso il file `stdio.h`.

La costante stringa `path` è una stringa di caratteri che identifica lo stream, ad esempio un file come ad esempio "`data.dat`" o "`input.dat`". La costante stringa `mode` è una stringa di caratteri (*flags*) che fornisce indicazioni sull'apertura dello stream. Le *flags* principali sono:

Flag	Modo	Operazione
`r`	*read*	Apre il file in lettura. Il file deve esistere. Lo stream è posizionato all'inizio.
`w`	*write*	Apre il file in scrittura. Se il file esiste il suo contenuto viene cancellato. Se il file non esiste viene creato. Lo stream è posizionato all'inizio.
`a`	*append*	Apre il file scrittura. Se il file non esiste viene creato. Lo stream è posizionato alla fine

Lo stream può essere anche aperto in modo *update*, ossia simultaneamente in scrittura ed in lettura simultaneamente, aggiungendo il carattere "`+`" ad una delle precedenti flags, in questo caso tuttavia per effettuare le operazioni di I/O è necessario spostarsi "manualmente" lungo lo stream utilizzando le funzioni `fsetpos()`, `fseek()`, `rewind()` o `fflush()`.

Tutti i modi, sia con che senza "+", possono essere seguiti dal carattere "b" per indicare che il file è binario. Ad esempio se `mode` vale "rb" lo stream binario è aperto in lettura. Nel caso di apertura in modo update "b" può sia precedere che seguire il carattere "+". Nel C89 Amendment1 e C99 gli streams aperti dalla funzione `fopen()` non hanno una caratteristica precisa, in altre parole possono essere utilizzati sia come stream di testo che come stream binari, anche se non contemporaneamente. Se uno stream è di testo o binario è determinato dalla prima operazione di Input/Output effettuata. In questi Standards la flag "b" non ha nessuno effetto ed è mantenuta solo per compatibilità con lo Standard C89.

Standard C permette di inserire altre informazioni nella stringa `mode`, ad esempio per specificare altri attributi dello stream, queste però in genere dipendono dal sistema.

Se l'apertura del file avviene senza errori la funzione `fopen()` restituisce il valore dell'indirizzo di memoria associato dell'oggetto di tipo `FILE` associato allo stream `path`. In caso di errore la funzione ritorna il puntatore nullo `NULL` ed assegna un valore alla variabile di errore `errno` in modo da poter risalire al tipo di errore occorso. La variabile `errno` è la variabile standard che in caso di errore contiene il *codice numerico* che identifica il tipo di errore. Tutti i codici di errore sono interi positivi il cui significato può dipendere dal sistema. I codici sono definiti nel file di header standard `errno.h` che va quindi incluso se si vuole utilizzare la variabile `errno` per risalire al tipo di errore.

Funzioni strerror() e perror()

Per conoscere il significato di un codice di errore si può utilizzare la funzione delle librerie standard

```
#include <string.h>

char *strerror(int errnum);
```

che prende come argomento il codice numerico di errore e restituisce (un puntatore a) una stringa con il messaggio di errore. Le seguenti istruzioni mostrano come aprire un file controllando che non vi siano errori utilizzando la funzione `strerror()`

```
FILE *in_file;

in_file = fopen("input.dat", "r");
if (in_file == NULL) {
  printf("Errore apertura 'input.dat': %s\n.", strerror(errno));
  exit(errno);
}
```

La funzione `printf()` utilizza la direttiva di conversione "`%s`" per scrivere la stringa di caratteri restitutita dalla funzione `strerror()`. Parleremo più dettagliatamente delle direttive di conversione tra breve.

Alternativamente si può utilizzare la funzione delle librerie standard

```
#include <stdio.h>

void perror(const char *string);
```

che prende come argomento una stringa `string`. La funzione `perror()` scrive sullo stream standard di errore `stderr` nell'ordine: la stringa `string`, il carattere ":", uno spazio bianco, un messaggio associato al codice di errore contenuto in `errno` ed un newline. Se `string` è il puntatore nullo `NULL` viene scritto solo il messaggio di errore.

Se si utilizza la funzione `perror()` al posto delle precedenti istruzioni avremo

```
FILE *in_file;

in_file = fopen("input.dat", "r");
if (in_file == NULL) {
  perorr("Errore apertura 'input.dat'");
  exit(errno);
}
```

In entrambi i casi se si verifica un errore nell'apertura del file `input.dat` verrà scritto il messaggio di errore associato al codice di errore contenuto nella variabile `errno` e chiamata la funzione `exit()` che interrompe l'esecuzione del programma restituendo il valore di `errno`.

Funzione exit()

La funzione delle librerie standard

```
#include <stdlib.h>

void exit(int status);
```

viene utilizzata ogni qual volta si debba interrompere l'esecuzione di un programma. La funzione permette di specificare un codice numerico `status` che viene restituito dal programma per fornire informazioni sul motivo dell'interruzione. Per convenzione in molti sistemi il valore `0` di `status` indica che il programma è terminato senza errori, mentre un valore non nullo indica qualche problema. Lo Standard C fornisce le due macros `EXIT_SUCCESS` ed `EXIT_FAILURE`, usualmente uguali a `0` e `1`, per indicare che il programma è stato terminato con successo o con qualche problema.

La funzione `exit()` non è analoga all'istruzione `return` poiché interrompe immediatamente l'esecuzione del programma e non della funzione attualmente in esecuzione. In altre parole l'esecuzione di qualsiasi funzione in sospeso viene interrotta. È possibile fare eseguire una o più funzioni anche dopo la chiamata della funzione `exit()` utilizzando la funzione `atexit()` per registrare le funzioni da eseguire prima di interrompere l'esecuzione del programma.

Funzione fclose()

Quando uno stream non è più necessario è possibile chiuderlo con la funzione

```
#include <stdio.h>

int fclose(FILE *stream);
```

dove `stream` è l'identificatore di uno stream aperto, ad esempio:

```
FILE *in_file;

in_file = fopen("input.dat", "r");
....
fclose(in_file);
```

La funzione `fclose()` chiude in modo ordinato lo stream svuotando prima della chiusura tutti i buffers associati allo stream. Se la chiusura non presenta errori la funzione ritorna il valore `0`, altrimenti ritorna il valore `EOF` e contemporaneamente assegna il codice di errore alla variabile `errno`. Il valore di `EOF` è un numero intero negativo associato a "nessun carattere" utilizzato per indicare la fine di un file *end-of-file*: quando si raggiunge la fine di uno stream non vi sono più caratteri da leggere. Il valore di `EOF` è definito nel file `stdio.h`. Se lo stream associato a `stream` è già stato chiuso il comportamento di `fclose()` può risultare arbitrario.

Gli streams sono chiusi anche dalla funzione `exit()`. Quando il programma incontra questa funzione l'esecuzione viene terminata svuotando tutti i buffers di I/O e chiudendo tutti gli streams aperti.

Se invece l'esecuzione del programma è causata dalla funzione

```
#include <stdlib.h>

void abort(void);
```

che produce una fine "anormale" del programma, i buffers non sono necessariamente svuotati prima della loro chiusura. Il comportamento in questi casi dipende dal sistema.

Funzione freopen()

Un identificatore di stream può essere riassociato ad un nuovo stream, chiudendo ovviamente il vecchio, utilizzando la funzione

```
#include <stdio.h>

FILE *freopen(const char *path, const char *mode,
              FILE *stream);
```

Questa funzione prende come parametri un file `path`, un modo `mode` ed un identificatore di uno streams aperto `stream`. Per prima cosa la funzione `freopen()` chiude lo stream aperto `stream`, come la funzione `fclose()`, ignorando qualsiasi tipo di errore, e poi apre il nuovo file utilizzando `path` e `mode` in modo analogo alla funzione `fopen()` con la differenza che non viene creato un nuovo oggetto `FILE` in una nuova posizione di memoria, ma viene utilizzata quella già individuata da `stream`. In altre parole vengono solo modificate le informazioni contenute nell'oggetto `FILE` all'indirizzo di memoria indicato da `stream`. Se non vi sono errori la funzione `freopen()` restituisce il valore di `stream`, se invece la nuova apertura fallisce restituisce il puntatore nullo `NULL`. Nello Standard C89 Amendment 1 e C99 la lo stream aperto con la funzione `freopen()` può essere utilizzato sia come stream di testo che binario, esattamente come se fosse stato aperto con la funzione `fopen()`. La funzione `freopen()` può essere utilizzata con qualsiasi stream aperto, ma di solito viene usata per riassociare gli streams standard `stdin`, `stdout` e `stderr`.

Funzione fflush()

Nelle operazioni di Output con buffers i dati non sono scambiati direttamente con i files ma con i buffers associati agli streams. Quando i buffers sono "pieni" vengono svuotati automaticamente dal sistema passando i dati al file associato.

È possibile tuttavia svuotare il contenuto dei buffers associati ad un stream di output o update in ogni momento utilizzando la funzione

```
#include <stdio.h>

int fflush(FILE *stream);
```

dove `stream` è l'identificatore di uno stream aperto. Ad esempio l'istruzione

```
fflush(stdout);
```

svuota il contenuto dei buffers associati allo stream standard di output.

Lo stato dello stream `stream` non viene modificato. Se `stream` è il puntatore nullo la funzione `fflush()` svuota il contenuto dei buffers di tutti gli streams di output. Se l'operazione non presenta errori la funzione `fflush()` restituisce il valore `0` altrimenti viene restituito il valore `EOF` ed assegnato un codice di errore alla variabile `errno`. La funzione `fflush()` viene di solito utilizzata in condizioni particolari quando è necessario conoscere istantaneamente i dati scritti, ad esempio durante la ricerca di errori nel programma (fase di *debugging*). Infatti se l'esecuzione del programma termina in modo "anormale" il sistema non sempre riesce a svuotare tutti i buffers per cui i dati non trasferiti ai files vanno persi.

Funzione rewind()

A volte può essere necessario dover rileggere uno stream dall'inizio, ad esempio per rileggere i dati. Questo può essere ottenuto chiudendo e riaprendo lo stream con le funzioni `fclose()` e `fopen()` o alternativamente con la funzione

```
#include <stdio.h>

void rewind(FILE *stream);
```

che semplicemente riposiziona il puntatore che indica la posizione lungo lo stream di testo o binario `stream` all'inizio dello stream.

2.14.3. Output su streams di testo: funzioni fprintf(), printf() e sprintf()

Le funzioni principali di output sono le funzioni della famiglia `printf` (*print-format*) ed in particolare

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
```

La funzione `fprintf()` (*file-printf*) scrive sullo stream specificato dal suo primo parametro `stream` una stringa di caratteri generata a partire dalla stringa di controllo `format` e, se presenti, dai valori delle espressioni date come parametri successivi ed interpretati secondo le direttive di conversione specificate in `format`.

La funzione `printf()` differisce dalla funzione `fprintf()` per il solo fatto che l'output è inviato allo stream di standard output `stdout`. Di conseguenza l'istruzione

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

è perfettamente equivalente all'istruzione

```
fprintf(stdout,"Ciao\n");
```

La funzione `sprintf()` invia l'output non su uno stream ma sulla stringa di buffer `str` specificata dal suo primo parametro, aggiungendo il carattere nullo di fine stringa '`\0`' (`NUL`) alla fine. È cura del programmatore assicurarsi che nella stringa di buffer `str` vi sia abbastanza spazio per contenere tutti i caratteri specificati dalla stringa `format`, inclusi gli eventuali valori delle espressioni, ed il carattere di fine stringa aggiunto dalla funzione `sprintf()`.

Dopo ogni operazione di output sia la funzione `fprintf()` che `printf()` spostano il puntatore che indica la posizione lungo lo stream alla prima posizione subito dopo l'ultimo carattere scritto in modo che una successiva operazione di output sullo stream inizia a scrivere di seguito senza ricoprire quanto scritto in precedenza. Questo comportamento può essere modificato con la stringa di controllo, ad esempio inserendo caratteri tipo il *backspace*. La funzione `sprintf()` invece scrive ogni volta a partire dal primo carattere della stringa indicata da `str` ricoprendo quindi qualsiasi cosa vi stata scritta in precedenza.

Lo Standard C richiede che tutte queste funzioni ritornino in caso di errore "`EOF`" o, nel caso che non vi siano errori, il numero di caratteri inviati all'output. Nel caso della funzione `sprintf()` il conto non include il carattere di fine stringa. Il valore di `EOF` definito nel file di header `stdio.h` è usato convenzionalmente per indicare la fine di un file *end-of-file*. Usualmente a `EOF` viene assegnato il valore -1 anche se lo Standard C richiede solo che il suo valore sia un valore costante intero e negativo.

La forma generale di un'istruzione di output con le funzioni `printf()`, `fprintf()`, `sprintf()` è

```
count = printf(format, parameter_1, ...);
count = fprintf(file_handler, format, parameter_1, ...);
count = sprintf(string_handler, format, parameter_1, ...);
```

ovvero se non interessa il valore della funzione,

```
printf(format, parameter_1, ...);
fprintf(file_handler, format, parameter_1, ...);
sprintf(string_handler, format, parameter_1, ...);
```

dove `parameter_1`, `parameter_2`, etc. sono i parametri il cui valore, convertito secondo le direttive contenute nella stringa di formattazione `format`, sono inviati output.

Formato di Output

Il formato con cui i dati vengono scritti sullo stream viene specificato con la costante stringa di controllo `format` composta da zero o più caratteri alfanumerici, da caratteri speciali e da direttive conversione che indicano il formato o conversione dei valori dei parametri che seguono. I caratteri alfanumerici che non fanno parte di caratteri speciali o direttive di conversione sono semplicemente copiati sull'output.

Caratteri Speciali. I caratteri speciali sono usati per scrivere caratteri che non possono essere inseriti direttamente o che hanno un significato particolare, come ad esempio i doppi apici o *return*. I caratteri speciali sono introdotti con i caratteri di escape sia di tipo carattere che di tipo numerico. Qui di seguito ricordiamo alcuni dei caratteri di escape maggiormente utilizzati:

Escape	Carattere	Nome
`\0`	`NUL`	carattere nullo
`\a`	`BEL`	alert (BEL)
`\b`	`BS`	backspace
`\t`	`HT`	tab orizzontale
`\n`	`NL`	newline o line feed
`\v`	`VT`	tab verticale
`\f`	`FF`	form feed
`\r`	`CR`	return
`\"`	`"`	doppi apici
`\\`	`\`	backslash
`\ONNN`		carattere con codice ottale `NNN`
`\xNN`		carattere con codice esadecimale `NN`

Tutti i caratteri di escape sono sostituiti con il carattere corrispondente prima di essere inviati all'output. Ad esempio il carattere speciale '`\n`' viene sostituito con un *newline* cosicché l'output continuerà su una nuova linea.

Direttive di conversione. Le direttive di conversione specificano come i valori delle espressioni fornite come parametri delle funzioni debbano essere interpretati e trasformati in caratteri. Tutte le direttive di conversione iniziano con il carattere '`%`' e sono seguite da una serie di caratteri che specificano oltre alle direttive specifiche di conversione per il tipo, ad esempio il tipo `int`, anche informazioni sul *prefisso* da aggiungere prima del valore, il numero di caratteri da utilizzare, la precisione numerica ed infine il *padding* ossia l'allineamento ed eventuali spazi bianchi o digit `0` per raggiungere il numero di caratteri richiesto.

Una stringa di formato può contenere più direttive di conversione che vengono applicate nell'ordine con cui compaiono nella stringa per i vari parametri della funzione: la prima direttiva per `parameter_1`, la seconda per `parameter_2`, e così via. Il tipo di ciascun parametro deve corrispondere correttamente alle direttive di conversione corrispondenti altrimenti in

generale il risultato non è prevedibile. Se il numero delle direttive è inferiore al numero di parametri viene segnalato un errore nel caso contrario, ossia se vi sono più direttive che parametri, le direttive in più vengono ignorate senza che sia prodotto nessun errore anche se il compilatore può segnalare la cosa con un messaggio di "warning".

Ogni direttiva di conversione inizia con il carattere "`%`" (percentuale) seguito nell'ordine da:

1. uno o più *flags* opzionali che modificano il significato delle conversioni:

flag	significato
`-`	allineamento a sinistra
`0`	usa `0` invece dello spazio per il *padding*
`+`	scrive il segno `+` o `-` se il valore è con segno
spazio	produce uno spazio o il segno `-`
`#`	introduce una variante del tipo di conversione

2. uno specificatore opzionale della *dimensione minima* del campo dato da un intero positivo indica il numero di caratteri minimi da scrivere.

3. uno specificatore opzionale della *precisione* dato dal punto decimale seguito da un intero positivo che indica il numero di cifre.

4. uno specificatore opzionale della *dimensione* dato da una delle seguenti lettere:

   `hh` La seguente conversione si applica ad un `signed char` o `unsigned char`.

   `h` La seguente conversione si applica ad un `short int` o `unsigned short int`.

   `l` La seguente conversione si applica ad un `long int` o `unsigned long int`.

   `L` La seguente conversione `e`, `E`, `f`, `F`, `g` o `G` si applica ad un `long double`.

5. uno specificatore *obbligatorio* di conversione espresso da uno dei seguenti caratteri:

   `d`, `i` Il parametro di tipo `int` è convertito in notazione decimale con segno. La precisione, se presente, fornisce il numero minimo di digits che devono essere scritti; se il valore contiene meno digits viene allineato a destra aggiungendo `0`. La precisione di default è `1`. Se il valore è negativo viene aggiunto il segno meno come prefisso. Se il valore è non-negativo ed è specificata il flag '`+`' il valore viene preceduto dal segno più. Se invece il valore è non-negativo ed è specificata il flag *space*, ma non il flag '`+`', il valore viene preceduto da uno spazio bianco. Se non è specificata nessun flag ed il valore è non-negativo non viene aggiunto nessun prefisso. Il flag '`#`' non è rilevante per queste conversioni.

`o`, `u`, `x`, `X` Il parametro di tipo `unsigned int` è convertito in notazione ottale senza segno (`o`), notazione decimale senza segno (`u`) o esadecimale senza segno (`x` o `X`). I caratteri `abcdef` sono usati con la conversione `x` mentre `ABCDEF` con la conversione `X`. La precisione, se presente, fornisce il numero minimo di digits che devono essere scritti; se il valore contiene meno digits viene allineato a destra aggiungendo `0`. La precisione di default è `1`. Le flags *space* e '`+`' non sono rilevanti per queste conversioni. Se viene specificato il flag '`#`' con la conversione `o` il valore ottale viene preceduto dal prefisso '`0`'. Se il flag '`#`' è presente con la conversione `x` il valore esadecimale viene preceduto dal prefisso '`0x`', o '`0X`' se la conversione è `X`. Il flag '`#`' non è rilevante per la conversione `u`.

`e`, `E` Il parametro di tipo `double` è arrotondato e convertito nel formato `[-]d.ddde±dd`. Il numero di digits dopo il punto decimale è uguale alla precisione, nel caso questa non sia specificata si usano 6 digits. Nella conversione `E` viene usata la lettera `E` per l'esponente. In numero di digits usati per l'esponente è lo stesso per tutti i valori ed è uguale al numero di digit necessario per rappresentare l'intervallo di valori del tipo floating-point. Se la precisione è `0` non vengono scritti digits dopo il punto decimale, inoltre questo non viene scritto a meno che non sia specificato il flag '`#`'. I flags *space* e '`+`' si comportano come nella conversione `d` o `i`.

`f`, `F` Il parametro di tipo `double` è arrotondato e convertito nel formato `[-]dddd.ddd`, dove il numero di digits dopo il punto decimale è uguale alla precisione. Nel caso questa non sia specificata si usano 6 digits. Se la precisione è esplicitamente zero il punto decimale non viene scritto. Vi è sempre almeno un digit prima del punto decimale. Se la precisione è `0` non vengono scritti digits dopo il punto decimale, inoltre questo non viene scritto a meno che non sia specificato il flag '`#`'. I flags *space* e '`+`' si comportano come nella conversione `d` o `i`.

`g`, `G` Il parametro di tipo `double` è convertito usando la conversione `f` (`F`) o `e` (`E`). La precisione indica il numero massimo di digits significativi, ossia diversi da `0`, dopo il punto decimale. Se non specificata si usano 6 digits, se invece è zero si assume il valore `1`. Nello Standard C la conversione `e` è usata se l'esponente del valore convertito è minore di -4 o se è maggiore o uguale alla precisione. Il valore viene ulteriormente modificato eliminando tutti gli zeri più a destra dopo il punto decimale. Se il risultato non ha zeri dopo il punto decimale questo non viene scritto. Se viene specificato il flag '`#`' gli zeri non vengono eliminati ed il punto decimale viene sempre scritto. I flags *space* e '`+`' si comportano come nella conversione `d` o `i`.

`c` Il parametro di tipo `int` è convertito in tipo `unsigned char` ed il corrispondente carattere viene scritto.

`s` Il parametro di tipo `const char *` deve contenere l'indirizzo di una stringa di caratteri (puntatore ad una stringa). Se la precisione non è specificata i caratteri della stringa sono scritti fino al carattere nullo '`\0`' che indica la fine della stringa escluso. Se la precisione è specificata sono scritti un numero di caratteri non più grande del valore della precisione. I flags *space*, '`+`' e '`#`' non sono rilevanti per la conversione `s`.

`p` Il parametro di tipo `void *` (puntatore generico) viene scritto in formato esadecimale con il prefisso `0x`. Su molti sistemi la conversione `p` è analoga alla conversione `o`, `x` o `X`. Lo specificatore di conversione `p` è Standard C ma non è necessariamente presente in altri dialetti.

`%` Viene scritto il carattere "`%`", non viene effettuata nessuna conversione. La forma completa per indicare il carattere percento è quindi "`%%`".

Lo specificatore di conversione obbligatorio termina la direttiva di conversione.

Lo Standard C89 Amendment 1 e il C99 hanno introdotto alcune modifiche alle direttive di output e nuovi specificatori di conversione, come ad esempio lo specificatore `a` (C99), dettate principalmente dalla necessità di trattare i nuovi tipi. Una delle modifiche introdotte dal C99 è l'uso dello specificatore di dimensione "`l`" (elle) che nel C99 può essere utilizzato anche per i tipi float e quindi con gli specificatori di conversione `f`, `e` e `g`, anche se essendo il valore di un argomento di tipo `float` sempre convertito a tipo `double` la presenza dello specificatore "`l`" è del tutto irrilevante. Nel C89 e C89 Amendment 1 invece lo specificatore di dimensione "`l`" può essere utilizzato in una direttiva di conversione di output solo con tipi interi. Per motivi di spazio non discuteremo oltre le modifiche ed aggiunte introdotte dal C89 Amendment 1 e C99, è bene però ricordarsi che vi possono essere differenze nelle direttive di output a seconda dello Standard utilizzato.

Di seguito sono riportati alcuni esempi di direttive di conversione semplici:

Conversione	**Tipo**
`%hd`	`signed short int`
`%d`	`signed int`
`%ld`	`signed long int`
`%hu`	`unsigned short int`
`%u`	`unsigned int`
`%lu`	`unsigned long int`
`%f`	`double`
`%Lf`	`long double`
`%e`	`double`

Il seguente programma illustra l'uso di alcuni dei campi opzionali nelle direttive di output di valori numerici interi e floating-point.

Programma: `print_format.c`

```c
#include <stdio.h>

int main(void)
{
  int int_n  = 123;
  double float_n = 0.123456789;

  printf("int    format %%d       : \"%d\"\n",         int_n);
  printf("int    format %%05d     : \"%05d\"\n",       int_n);
  printf("int    format %%7d      : \"%7d\"\n",        int_n);
  printf("int    format %%-7d     : \"%-7d\"\n",       int_n);
  printf("int    format %%7.4d    : \"%7.4d\"\n",      int_n);
  printf("int    format %%-7.4d   : \"%-7.4d\"\n",     int_n);
  printf("double format %%f       : \"%f\"\n",       float_n);
  printf("double format %%10.5f   : \"%10.5f\"\n",   float_n);
  printf("double format %%-12.5e  : \"%-12.5e\"\n", float_n);
  return(0);
}
```

Quando questo programma viene eseguito si ha il seguente output:

```
int    format %d       : "123"
int    format %05d     : "00123"
int    format %7d      : "    123"
int    format %-7d     : "123    "
int    format %7.4d    : "   0123"
int    format %-7.4d   : "0123   "
double format %f       : "0.123457"
double format %10.5f   : "   0.12346"
double format %-12.5e  : "1.23457e-01 "
```

infatti

Conversione	**Variabile**	**Output**	**Note**
%d	integer	”123”	Dimensione del campo 3
%05d	integer	”00123”	Dimensione del campo 5, allineato a destra, padding 0
%7d	integer	”␣␣␣␣123”	Dimensione del campo 7, allineato a destra, padding *spazio*
%-7d	integer	”123␣␣␣␣”	Dimensione del campo 7, allineato a sinistra
%7.4d	integer	”␣␣␣0123”	Dimensione del campo 7, allineato a destra, 4 quattro digits
%-7.4d	integer	”0123␣␣␣”	Dimensione del campo 7, allineato a sinistra, 4 quattro digits
%f	float	”0.1234567”	Dimensione campo 8, precisione default 6 , allineato a destra

Conversione	Variabile	Output	Note
%10.5f	float	"␣␣␣0.12346"	Dimensione del campo 10, precisione 5, allineato a destra
%-12.5e	float	"1.23457e−01␣␣"	Dimensione del campo 12, precisione 5, allineato a sinistra

In questo esempio i doppi apici sono stati messi solo per evidenziare la dimensione del campo.

2.14.4. Input da streams di testo: funzioni fscanf(), scanf() e sscanf()

Generalmente per le operazioni di input da stream di testo si utilizzano le funzioni della famiglia `scanf` (*scan-format*), ed in particolare

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

La funzione `fscanf()` legge un input formattato dallo stream di input `stream` specificato dal suo primo parametro, ne interpreta i caratteri secondo le direttive di conversione specificate dalla stringa di controllo `format` e scrive il risultato nelle locazioni di memoria associate alle variabili specificate dai parametri successivi della funzione. Tutti i parametri dopo la stringa di controllo devono quindi contenere indirizzi di memoria di variabili (*puntatori*).

La lettura dei caratteri di input si ferma quando sono state esaurite tutte le direttive di conversione specificate nella stringa `format`, oppure quando la conversione non può essere effettuata.

La funzione `scanf()` differisce dalla funzione `fscanf()` per il solo fatto che l'input è preso dallo stream standard di input `stdin`, mentre la funzione `sscanf()` legge i caratteri dalla stringa `str` specificata dal suo primo parametro e non da uno stream di testo.

Dopo ogni operazione di input sia la funzione `fscanf()` che `scanf()` spostano il puntatore che indica la posizione lungo lo stream alla prima posizione subito dopo l'ultimo carattere letto in modo che una successiva operazione di input dallo stream legge i caratteri seguenti. Nel caso lo stream standard di input `stdin` sia associato alla tastiera il puntatore viene spostato all'inizio del prossimo input da tastiera. La funzione `sscanf()` invece inizia ogni operazione di input sempre dal primo carattere della stringa indicata da `str`. Di conseguenza mentre è possibile leggere da uno stream il valore di due variabili con due chiamate successive della funzione `fprintf()` questo non è possibile da una stringa. I due valori vanno letti con una sola chiamata della funzione `sscanf()` poiché chiamate successive della funzione leggerebbero sempre il stesso valore, il primo, come mostra il seguente programma

Programma: `sscanf.c`

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

```
int main(void)
{
  int i,j;
  char str[10];

  /* La sringa contiene i valori 10 e 20 */
  sprintf(str, "%d %d", 10, 20);

  sscanf(str, "%d", &i);
  sscanf(str, "%d", &j);

  printf("successive: i = %d j = %d\n", i, j);

  sscanf(str, "%d%d", &i, &j);

  printf("unica     : i = %d j = %d\n", i, j);

  return 0;
}
```

Quando si esegue il programma si ha infatti:

```
successive: i = 10 j = 10
unica     : i = 10 j = 20
```

da cui si vede chiaramente che le due chiamate successive della funzione `sscanf()` hanno letto dalla stringa `str` sempre lo stesso valore.

Le funzioni `scanf()`, `fscanf()` e `scanf()` ritornano come valore un intero uguale al numero di conversioni ed assegnazioni fatte che può essere uguale o minore, nel caso vi sia un errore di conversione o assegnazione, al numero di assegnazioni richieste in `format`. Se l'errore, o la fine dello stream di input, avviene dopo che siano state fatte delle conversioni ed assegnazioni il valore restituito dalle funzioni è uguale al numero di conversioni ed assegnazioni effettuate correttamente. Se invece l'errore avviene prima di iniziare le conversioni, o se non vi è nulla nello stream di input, viene restituito il valore di end-of-file `EOF`. Il valore può essere anche uguale a `0` se vi è uno stream di input disponibile non è stata effettuata nessuna conversione ed assegnazione. Di solito questo è dovuto alla presenza in input di uno o più caratteri che non possono essere convertiti con la direttiva richiesta, ad esempio un carattere alfabetico con una direttiva di tipo "`%d`".

La forma generale di un'istruzione di input con le funzioni `fscanf()`, `scanf()` e `sscanf()` è

```
count = scanf(format, &parameter_1, ...);
count = fscanf(file_handler, format, &parameter_1, ...);
count = sscanf(string_handler, format, &parameter_1, ...);
```

ovvero se il numero di conversioni ed assegnazioni effettivamente fatte non interessa,

```
scanf(format, &parameter_1, &parameter_2, ...);
fscanf(file_handler, format, &parameter_1, ...);
sscanf(string_handler, format, &parameter_1, ...);
```

Osserviamo che sebbene la sintassi delle funzioni della famiglia `printf` e quelle della famiglia `scanf` siano molto simili vi è una differenza fondamentale che spesso è causa di errore. Nelle funzioni di output la lista delle variabili da convertire è formata dagli identificatori delle variabili di cui si vuole scrivere il valore. Nelle funzioni di input, invece, la lista è si formata dagli identificatori delle variabili a cui si vogliono assegnare i valori letti ma questi sono *preceduti* dal carattere "`&`" (*ampersand*). Questo carattere indica l'operatore unario `&` il cui valore è l'indirizzo di memoria del suo operando, in questo caso delle variabili `parameter_1`, `parameter_2`, etc.. Questa differenza è dovuta al fatto che le funzioni di input per poter scrivere i valori letti nella memoria associata alle variabili prendono come argomenti gli indirizzi di memoria (puntatori) delle variabili e non le variabili stesse. Di conseguenza tutti i parametri che seguono la stringa di controllo `format` devono essere del tipo `(T *)` dove `T` è il tipo della variabile a cui vanno assegnati valori letti. Questa notazione diventerà chiara più avanti discutendo le funzioni ed i puntatori.

Siccome le funzioni di input interpretano il valore dei parametri associati alle variabili come indirizzi di memoria, se non si mette l'operatore `&` di fronte al nome delle variabili il risultato non è prevedibile. Nel migliore dei casi si ottiene un messaggio di errore del tipo "Illegal memory access" o "Segmentation violation core dumped" o ancora "Segmentation fault core dumped". In altri casi, meno fortunati, può accadere che venga cambiato il valore di una variabile a caso diversa da quella voluta, con risultati a volte disastrosi. Questi errori possono essere piuttosto difficili da individuare. Su sistemi UNIX il danno è comunque limitato poiché vi sono dei sistemi per limitare l'effetto di questi danni. Su altri sistemi, ed in particolare su MS-DOS/Windows, che non hanno protezioni della memoria i danni possono essere tali da causare un crash dell'intero sistema.

Formato di Input

Come le funzioni della famiglia `printf` anche quelle della famiglia `scanf` utilizzano una stringa di caratteri per specificare il formato di conversione. Nel caso delle funzioni di input la stringa di controllo `format` contiene le `direttive` di conversione per trasformare i caratteri letti dall'input in valori da assegnare alle variabili specificate dai parametri che seguono la stringa. La stringa può contenere più direttive di conversione che sono eseguite in sequenza assegnando il valori alle variabili nell'ordine con cui sono date dopo la stringa: prima direttiva prima variabile, seconda direttiva seconda variabile e così via. Il tipo delle variabili che seguono la stringa `format` deve corrispondere correttamente alle direttive di conversione contenute nella stringa altrimenti il risultato non è prevedibile.

Ogni direttiva di conversione inizia con il carattere `%` e finisce con uno specificatore di formato.

La direttiva di conversione può includere subito dopo `%` un campo opzionale che indica la dimensione del campo da leggere. Questo è un intero positivo che indica il numero massimo di caratteri da leggere dall'input. Se la dimensione non è specificata in genere vengono letti tutti i caratteri fino al primo *space*, *tab*, *newline* o *return* e convertiti secondo la direttiva di conversione specificata. Ad esempio la direttiva di conversione "`%2d`" indica di leggere 2 caratteri e di convertirli in `signed int` ("`d`"). Quindi se ad esempio lo stream di input contiene "1234" solo i primi due caratteri "12" saranno letti. Se invece la dimensione del campo è omessa, ossia si usa "`%d`", tutti e quattro i caratteri "1234" verranno letti.

Gli specificatori di formato sono:

`d` Conversione tipo intero decimale con segno. Il corrispondente parametro deve essere di tipo `int *`. Il valore da leggere deve essere espresso in notazione decimale, ossia come una serie di digits decimali eventualmente preceduti dal segno opzionale '`+`' o '`-`'. Se il valore è troppo grande per essere rappresentato il risultato è indefinito.

`i` Conversione tipo intero con segno. Il corrispondente parametro deve essere di tipo `int *`. Il valore è letto in formato ottale se inizia con il carattere '`0`', in formato esadecimale se inizia con i caratteri '`0x`' o '`0X`' o decimale negli altri casi. Il valore deve essere espresso con una serie di digits della rappresentazione corrispondente preceduti eventualmente dal segno opzionale '`+`' o '`-`'. Se il valore è troppo grande per essere rappresentato il risultato è indefinito.

`u` Conversione tipo intero decimale senza segno. Il corrispondente parametro deve essere di tipo `unsigned int *`. Il valore da leggere deve essere espresso in notazione decimale e può essere preceduto dal segno '`+`' o '`-`'. Se il valore è troppo grande per essere rappresentato il risultato è indefinito.

`o` Conversione tipo intero ottale senza segno. Il corrispondente parametro deve essere di tipo `unsigned int*`. Il valore da leggere deve essere espresso in notazione ottale e può essere preceduto dal segno '`+`' o '`-`'. Se il valore è troppo grande per essere rappresentato il risultato non è prevedibile.

`x` Conversione tipo intero esadecimale senza segno. Il corrispondente parametro deve essere di tipo `unsigned int *`. Il valore da leggere deve essere espresso in notazione esadecimale e può essere preceduto dal segno '`+`' o '`-`'. Sono permessi sia i caratteri minuscoli `abcdef` che maiuscoli `ABCDEF`. Se il valore è troppo grande per essere rappresentato il risultato non è prevedibile. Alcuni sistemi accettano anche lo specificatore `X` come equivalente a `x`, questo però è un'estensione non Standard C.

`g`, `e`, `f` Conversione tipo decimale floating-point con segno. Il corrispondente parametro deve essere di tipo `float *`. Il valore da leggere deve essere in formato decimale con o senza punto decimale e con o senza esponente specificato dal carattere `e` o `E` seguito da digits decimali con o senza segno. Il valore può essere preceduto dal segno opzionale '`+`' o '`-`'. Se il valore non può essere rappresentato esattamente viene arrotondato o troncato. Se il valore è troppo grande o troppo piccolo il risultato può dipendere dal sistema anche se lo Standard C richiede che venga restituito il valore `HUGE_VAL`. Le conversioni `g`, `e` e `f` sono perfettamente equivalenti. Alcuni sistemi accettano anche le conversioni `G`, `E` e `F`, però questa estensione non è Standard C.

`s` Conversione tipo stringa. Viene letta una sequenza di caratteri non-bianchi, la lettura si ferma quindi quando viene letto un carattere bianco *space*, `tab`, `newline` o `return`. Se viene specificata la dimensione del campo la lettura si ferma quando sono stati letti un numero di caratteri uguali alla dimensione specificata o al primo carattere bianco, a seconda di cosa avviene prima. La conversione `s` aggiunge dopo l'ultimo carattere letto il carattere nullo '`\0`' di fine stringa. Il corrispondente parametro deve essere di tipo `char *` e deve contenere l'indirizzo di una zona di memoria di dimensione sufficiente a contenere tutti i caratteri letti più il carattere nullo. Nella conversione `s` eventuali caratteri bianchi all'inizio vengono ignorati.

`c` Conversione tipo carattere. Uno o più caratteri sono letti a seconda della dimensione del campo specificata. Se non viene specificata nessuna dimensione la dimensione viene presa uguale a `1` e viene letto un solo carattere. Il corrispondente parametro deve essere di tipo `char *` e deve contenere l'indirizzo di una zona di memoria di dimensione sufficiente a contenere tutti i caratteri letti. La conversione `c` non aggiunge il carattere nullo '`\0`' ne salta gli eventuali spazi iniziali.

`p` Conversione tipo puntatore. Il valore letto viene interpretato come un indirizzo di memoria. Il parametro corrispondente deve essere del tipo `void **`. Il formato del valore dipende dal sistema, anche se su molti sistemi è lo stesso di quello prodotto dalla direttiva di conversione "`%p`" della funzione `printf()`.

`n` Non viene effettuata nessuna conversione e non vengono letti caratteri. Al parametro corrispondente, che deve essere di tipo `int *`, viene assegnato come valore il numero di caratteri letti fino a quel momento.

Lo specificatore di formato può essere preceduto da uno specificatore di lunghezza `h`, `l` o `L` che indica le dimensioni del tipo in cui deve essere convertita la sequenza di caratteri letti dall'input:

Specificatore	Conversione
`h`	conversione a `short int` (invece che `int`) con formato `dioux`
`l`	conversione a `long int` (invece che `int`) con formato `dioux` o conversione a `double` (invece che `float`) con formato `efg`
`ll` o `L`	conversione a `long double` (invece che `float`) con formato `efg`

Lo Standard C89 Amendment 1 e il C99 hanno introdotto alcune modifiche alle direttive di conversione tra cui nuovi specificatori di conversione. Questi non verranno discussi in questo Primer.

La stringa di formato può contenere anche altri caratteri. Spazi bianchi come *space*, *tab* o *newline* nella stringa di controllo `format` vengono associati con un qualsiasi numero, anche zero, di spazi bianchi nell'input. Quindi le stringhe di formato "`%d%d`", "`%d %d`", "`%d\t%d`" o "`%d\n%d`" sono tutte equivalenti. Ogni altro carattere nella stringa di formato è associato solo con se stesso, questo vuol dire che lo stream di input dovrà contenere *esattamente* gli stessi caratteri nelle posizioni indicate da `format`. Il seguente programma chiarisce questo punto.

Programma: `input_format.c`

```
#include <stdio.h>

int main(void)
{
  int  a, b;
  int count;

  printf("[a] word [b]: ");
  count = scanf("%1d word %2d", &a, &b);
  printf("a: %d\n", a);
  printf("b: %d\n", b);
  printf("count -> %d\n", count);
  return 0;
}
```

In questo esempio la stringa di controllo è "`%1d word %2d`" per cui lo stream di input dovrà essere formato esattamente da un digit decimale ("`%1d`"), zero o più caratteri bianchi *space* o *tab* o *newline*, i caratteri "`word`" ("`word`") e da uno più digits decimali ("`%2d`"). Osserviamo che mentre il numero prima della parola `word` deve essere formato esattamente da un digit, quello dopo `word` può essere formato da un numero arbitrario di digits, in ogni caso solo i primi due saranno letti ("`%2d`"). Questa differenza di comportamento è dovuta al fatto che nella stringa di formato viene specificato che dopo il primo numero di un digt vi è la parola `word` mentre non vengono date indicazioni su quello che deve seguire il secondo numero, e quindi può esservi qualsiasi cosa anche più cifre di quelle richieste.

Di seguito sono riportati alcuni esempi di utilizzo corretto o errato del programma.

- `[a] word [b]: 1 word 23`

```
a: 1
b: 23
count -> 2
```

Dopo `word` vi sono due digit che vengono letti. Il contatore indica che sono state effettuate due conversioni.

- `[a] word [b]: 1word23`
  ```
  a: 1
  b: 23
  count -> 2
  ```

Gli spazi bianchi nello stream di input non contano. Lo stesso vale per il *newline*

```
[a] word [b]: 1
word 23
a: 1
b: 23
count -> 2
```

che è trattato come uno spazio bianco.

- `[a] word [b]: 1 word 2`
  ```
  a: 1
  b: 2
  count -> 2
  ```

Dopo `word` vi è un solo digit che viene letto poiché la dimensione massima del campo è 2. Il contatore di nuovo indica che sono state effettuate due conversioni.

- `[a] word [b]: 1 word 12345`
  ```
  a: 1
  b: 12
  count -> 2
  ```

In questo caso dopo `word` vi sono cinque digits ma solo i primi due vengono letti poiché la dimensione massima del campo è 2. Il contatore di indica che sono state effettuate correttamente due conversioni.

- `[a] word [b]: 1 ward 23`
  ```
  a: 1
  b: 134518128
  count -> 1
  ```

In questo caso il programma produce un risultato errato perché lo stream di input contiene la parola `ward` invece di `word`. L'errore di conversione avviene dopo che è stata effettuata la prima conversione per cui il contatore vale `1` ed infatti il primo numero è letto correttamente.

- `[a] word [b]: 12 word 34`
  ```
  a: 1
  b: 134518128
  count -> 1
  ```

In questo caso il programma produce un risultato errato perché prima della parola `word` lo stream di input contiene due digits decimali mentre il formato ne richiede uno solo. Di nuovo l'errore di conversione avviene dopo che è stata effettuata la prima conversione utilizzando solo il primo dei due digits come mostrato dal valore del contatore.

- `[a] word [b]: word 23`
  `a: 134518384`
  `b: 134518128`
  `count -> 0`

In questo caso il programma produce un risultato errato poiché nello stream di input non vi sono digits prima della parola `word`. In questo caso l'errore di conversione avviene prima che sia stata effettuata qualsiasi conversione per cui il valore del contatore è `0`.

Il seguente programma che copia un file chiamato `source.dat` sul file `dest.dat` scambiando le due colonne del file sorgente illustra l'uso delle funzioni di Input/Output con files.

Programma: `copy.c`

```c
#include <stdio.h>

int main(void)
{
  int    i;
  double x;

  FILE *fp_in, *fp_out;

  fp_in = fopen("source.dat", "r");
  if (fp_in == NULL ) {
    perror("source.dat");
    return 1;
  }

  fp_out = fopen("dest.dat", "r");
  if (fp_out == NULL ) {
    perror("dest.dat");
    return 1;
  }

  while ( fscanf(fp_in, "%d %lf", &i, &x) == 2 ) {
    fprintf(fp_out, "%f\t%d\n", x, i);
  }

  fclose(fp_in);
  fclose(fp_out);

  return 0;
}
```

In questo esempio si assume che il file sorgente sia composto da due colonne, la prima con valori di tipo `int` e la seconda con valori di tipo `double`. Ad esempio

```
1    3.9
2    3.7
19   5.1
.......
```

Si noti che i valori di tipo `double` sono letti con la direttiva di conversione "`%lf`" e scritti con la direttiva di conversione"`%f`". Inoltre poiché non si conosce il numero delle linee del file sorgente le linee vengono lette e copiate fino a che non si raggiunge la fine del file o si ha un errore di lettura:

```
while ( fscanf(fp_in, "%d %lf", &i, &x) == 2 )
```

Le parentesi graffe nell'istruzione `while` non sono necessarie in quanto il corpo del ciclo è composto da una sola istruzione. Sono state utilizzate al solo scopo di facilitare la lettura del programma.

2.14.5. Input da stream di testo con buffer: funzione fgets()

La funzione `scanf()` è la "bestia nera" del linguaggio C poiché fornisce un modo di leggere dei dati che difficilmente funziona. Infatti la funzione è nota per la sua cattiva gestione degli end-of-line nella la divisione in linee dello stream di input da tastiera che ne rende l'uso spesso piuttosto difficoltoso. Il modo più semplice di risolvere questi problemi è quello di non usare la funzione `scanf()` per leggere streams di input da tastiera.

Fortunatamente in C spesso vi sono molti modi di risolvere lo stesso problema. In questo caso un metodo alternativo che elimina i problemi di gestione degli end-of-line è quello di leggere ciascuna linea dello stream di input da tastiera in un *buffer* temporaneo da cui poi estrarre solo le informazioni che servono togliendo così alla funzione il compito di dividere lo stream in linee. Questo può essere realizzato facilmente utilizzando la funzione

```
#include <stdio.h>

char *fgets(char *str, int size, FILE *stream);
```

per leggere tutta una linea dello stream di input e copiarla su una stringa di caratteri utilizzata come buffer e poi utilizzare la funzione `sscanf()` per estrarre dalla stringa i dati che interessano.

La funzione `fgets()` legge al massimo `size-1` caratteri dallo stream specificato dal suo terzo parametro `stream` e li scrive nella stringa specificata dal suo primo parametro `str`. Un carattere viene sempre riservato per carattere di fine stringa '\0' (`NUL`). La lettura si ferma quando viene letto un `EOF` o un newline (*end-of-line*) o se sono stati letti `size-1` caratteri. Se la lettura termina perché è stato letto un newline il carattere di newline '\n' viene scritto nella stringa. In ogni caso dopo l'ultimo carattere letto viene sempre aggiunto il carattere di fine stringa '\0'.

Se la lettura dallo stream termina senza problemi la funzione `fgets()` restituisce come valore l'indirizzo di memoria della stringa `str` (*puntatore*). Se non viene letto nulla la funzione resti-

tuisce il valore `NULL` (puntatore nullo) ed il contenuto della stringa `str` non viene modificato. Il valore `NULL` viene restituito anche nel caso di errore nell'operazione di input ma in questo caso il contenuto della stringa `str` non è prevedibile. La funzione `fgets()` non distingue tra un errore di input e la fine dello stream (*end-of-file*).

I seguenti due programmi mostrano i due metodi di input da tastiera a confronto. Supponiamo di voler leggere da tastiera un valore intero ed uno floating-point. Un modo è quello di utilizzare la funzione `scanf()` come mostrato nel seguente programma

Programma: `scanf.c`

```
#include <stdio.h>

int main(void)
{
   int value_i;
   double value_f;

   printf("Dammi un valore intero ed uno floating-point: ");
   scanf("%d%lf", &value_i, &value_f);

   printf("I valori letti sono -->  %d e %f\n", value_i, value_f);
   return(0);
}
```

Lo stessa operazione può essere effettuata utilizzando la funzione `fgets()` ed una stringa di buffer:

Programma: `fgets-sscanf.c`

```
#include <stdio.h>

int main(void)
{
   int value_i;
   double value_f;
   char line[81];

   printf("Dammi un valore intero ed uno floating-point: ");
   fgets(line, sizeof(line), stdin);
   sscanf(line,"%d%lf", &value_i, &value_f);

   printf("I valori letti sono -->  %d e %f\n", value_i, value_f);
   return(0);
}
```

In questo caso i dati dello stream di input sono prima letti nel buffer `line` che viene poi processato dalla funzione `sscanf()` per estrarre i valori letti e assegnarli alle variabili `value_i` e `value_f`. In questo esempio il buffer `line` può contenere fino ad un massimo di 80 caratteri poiché uno serve per il carattere '`\0`' di fine stringa. L'uso dell'operatore `sizeof` assicura che non vengano letti più caratteri di quanti il buffer possa contenerne.

2.14.6. Input/Output da streams binari

Le operazioni di Input/Output da streams binari sono effettuate mediante le funzioni

```
#include <stdio.h>

size_t  fread(void  *ptr,  size_t size, size_t nmemb,
              FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t  nmemb,
              FILE *stream);
```

La funzione `fread()` legge `nmemb` oggetti ciascuno di dimensione `size` dallo stream binario specificato dall'identificatore `stream` e li scrive nelle locazioni di memoria indicate dal suo primo parametro `ptr`. È cura del programmatore assicurarsi che la zona di memoria indicata da `ptr` sia sufficiente per memorizzare `nmemb` oggetti di dimensione `size`.

La funzione `fwrite()` fa l'operazione inversa ossia scrive sullo stream binario specificato dall'identificatore `stream` `nmemb` oggetti di dimensione `size` presi a partire dall'indirizzo di memoria specificato dal suo primo parametro.

Non appena avremo introdotto i puntatori risulterà chiaro che in entrambi i casi `ptr` è un puntatore ad un array di `nmemb` oggetti di ciascuno di dimensione `size`.

In queste funzioni compaiono due tipi nuovi: `size_t` e `void *`. Il primo, `size_t`, è un tipo intero privo di segno utilizzato per oggetti il cui valore sono dimensioni (*size type*). Di solito il tipo `size_t` è equivalente al tipo `unsigned int` o `unsigned long int` a seconda del sistema. Nello Standard C il tipo `size_t` è definito nel file di header `stddef.h` e, per convenienza, anche nel file di header `stdlib.h`.

Il tipo `void *` è un *puntatore generico*, ossia associato a nessun tipo specifico, il che permette di utilizzare questa funzione per leggere dallo stream binario di input dati di un qualsiasi tipo permesso dal C. Infatti se ad esempio il primo parametro della funzione `fread()` fosse stato del tipo `int *` la funzione avrebbe potuto leggere solo dati di tipo `int`, o meglio, la funzione avrebbe interpretato qualsiasi tipo di dati letti dallo stream binario come dati di tipo `int` e le li avrebbe scritti nelle locazioni di memoria indicate da `ptr` in questo formato. Utilizzando invece il puntatore generico il tipo di dati letti viene determinato dal tipo di puntatore dato come primo parametro alla funzione. Questi concetti saranno più chiari una volta introdotti i puntatori, per il momento è sufficiente sapere che se alla funzione viene dato come primo parametro un puntatore ad un oggetto di tipo `int`, ossia un indirizzo di memoria di un oggetto di tipo `int`, i dati letti saranno interpretati come dati di tipo `int`. Se invece viene fornito un puntatore ad un oggetto di tipo `double`, ossia l'indirizzo di memoria di un oggetto di tipo `double`, i dati saranno interpretati come dati di tipo `double`. Ad esempio per leggere da uno stream binario un valore intero ed assegnarlo ad una variabile di tipo `int` si userà un'istruzione del tipo

```
int      var;
FILE *stream;

fread(&var, 1, sizeof(int), stream);
```

mentre se si vuole leggere il valore floating-point ed assegnarla ad una variabile di tipo `double` si dovrà usare un'istruzione del tipo

```
double   var;
FILE *stream;

fread(&var, 1, sizeof(double), stream);
```

Ricordiamo che l'operatore unario `&` fornisce l'indirizzo della variabile. In entrambi i casi l'uso dell'operatore `sizeof` ci permette di fornire la dimensione corretta del tipo.

Nota. L'uso dell'operatore `sizeof` nella lettura di stream binari può causare errori se i files binari sono stati scritti su un sistema che utilizza dimensioni diverse per i tipi. In questi casi è necessario sapere esattamente quale è la dimensione con cui sono stati scritti i dati e rileggerli con questa dimensione. Come abbiamo già avuto modo di dire i files binari sono molto meno portabili da un sistema ad altro dei dei file di testo.

Il seguente programma che scrive sul terminale i caratteri letti da uno stream binario mostra un esempio dell'utilizzo della funzione `fread()`.

Programma: `fread.c`

```
#include <stdio.h>

int main(void)
{
  char     c;
  char s[80];

  FILE *fp;

  fp = fopen("testo.unix", "r");

  /* legge lo stream testo e scrive il contenuto */
  while( fgets(s, sizeof(s), fp) != NULL )
    printf("testo: %s", s);

  fclose(fp);

  printf("\n");

  /* legge lo stream binario e scrive il contenuto */
  fp = fopen("testo.unix", "rb");
  while( fread(&c, sizeof(c), 1, fp) != 0)
    printf("binario: %02X (%03d) '%c'\n", c, c, c);

   return 0;
}
```

Se il file `testo.unix` contiene le due righe

```
Prova
end
```

quando il programma viene eseguito sul terminale si ha

```
testo: Prova
testo: end

binario: 50 (080) 'P'
binario: 72 (114) 'r'
binario: 6F (111) 'o'
binario: 76 (118) 'v'
binario: 61 (097) 'a'
binario: 0A (010) '
'
binario: 65 (101) 'e'
binario: 6E (110) 'n'
binario: 64 (100) 'd'
binario: 0A (010) '
'
```

Si noti il newline, codice ASCII `0A`, alla fine di ogni linea. Se invece il file di testo fosse stato scritto su un sistema MS-DOS/Windows sullo schermo si sarebbe visto

```
testo: Prova
testo: end

binario: 50 (080) 'P'
binario: 72 (114) 'r'
binario: 6F (111) 'o'
binario: 76 (118) 'v'
binario: 61 (097) 'a'
'inario: 0D (013) '
binario: 0A (010) '
'
binario: 65 (101) 'e'
binario: 6E (110) 'n'
binario: 64 (100) 'd'
'inario: 0D (013) '
binario: 0A (010) '
'
```

perché il sistema MS-DOS/Windows usa come end-of-line due caratteri: `CR`, codice ASCII `0D`, e `NL`, codice ASCII `0A`.

2.15. Esempio: Interpolazione lineare di un set di dati (Rev. 2.1.1)

Come esempio di input di dati da un file consideriamo il seguente programma che calcola l'interpolazione (*fit*) lineare

$$y = a + bx$$

di un set di dati $[(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots]$ letti da un file. Chiaramente il programma dipende un parte dalle informazioni che si hanno a disposizione. Ad esempio il numero di coppie di valori nel file o il numero di coppie da utilizzare per il fit o l'intervallo di variabilità della variabile indipendente x da utilizzare e così via. Per rendere lo sviluppo del programma il più semplice possibile supporremo di non avere nessuna informazione aggiuntiva oltre al nome del file e che questo contiene coppie di valori `(x_i,y_i)`. Il programma può poi essere facilmente modificato per soddisfare qualsiasi altra richiesta.

Come per la scrittura di tutti i programmi la prima cosa da fare è di individuare un algoritmo che risolve il problema. Nel caso del fit lineare un algoritmo semplice per determinare il valore di a e b è ottenuto con il metodo dei minimi quadrati che stima i valori di a e b come quei valori per i quali la funzione di "scarto"

$$\chi^2(a,b) = \sum_{i=1}^{N} [y_i - a - bx_i]^2$$

assume il valore minimo possibile. Ponendo uguali a zero le derivate di $\chi^2(a,b)$ rispetto a a e b si ottiene un sistema di equazioni lineari nelle incognite a e b la cui soluzione è:

$$\begin{aligned} a &= \frac{1}{D}\left[S_{xx}S_y - S_x S_{xy}\right] \\ b &= \frac{1}{D}\left[S_{xy}S_1 - S_x S_y\right] \end{aligned}$$

dove

$$D = S_{xx}S_1 - {S_x}^2$$

e

$$\begin{aligned} S_x &= \sum_{i=1}^{N} x_i & S_y &= \sum_{i=1}^{N} y_i \\ S_{xx} &= \sum_{i=1}^{N} {x_i}^2 & S_{xy} &= \sum_{i=1}^{N} x_i\, y_i \end{aligned}$$

ed $S_1 = \sum_{i=1}^{N} 1 \equiv N$.

L'errore sui valori stimati di a e b è

$$\begin{aligned} {\sigma_a}^2 &= \frac{\chi^2(a,b)}{(N-2)}\frac{S_{xx}}{D} \\ {\sigma_b}^2 &= \frac{\chi^2(a,b)}{(N-2)}\frac{S_1}{D} \end{aligned}$$

dove $\chi(a,b)^2$ è il valore calcolato sui valori stimati. Sviluppando i quadrati $\chi(a,b)^2$ può essere facilmente scritto in termini delle somme già calcolate più la nuova somma

$$S_{yy} = \sum_{i=1}^{N} {y_i}^2$$

come

$$\chi(a,b)^2 = S_{yy} + a^2 S_1 + b^2 S_{xx} - 2aS_y - 2bS_{xy} + 2abS_x$$

e quindi facilmente valutato senza dover ricalcolare la somma che definisce $\chi(a,b)^2$ con i valori stimati di a e b.

Il seguente programma effettua il fit lineare utilizzando l'algoritmo dei minimi quadrati appena descritto.

Programma: `lin_fit.c`

```
/*
 * Descrizione : calcola il fit lineare
 *                   y = a + b x
 *               di un set di dati (x_i,y_i).
 *
 * Input       : nome del file contenete i dati
 *
 * Output      : su terminale # coppie lette, a, b ed errori.
 *
 * $yaC-Primer: lin_fit.c v 1.3 02.02.05 AC $
 */
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(void)
{
  int    sum_1;                    /* # coppie dati      */
  double x, y;                     /* coppie di dati     */
  double sum_x,  sum_y;            /* variabili cumulate */
  double sum_xx, sum_xy, sum_yy;   /* variabili cumulate */
  double coef_a, coef_b, det;      /* coefficienti retta */
  double sigma_a, sigma_b;         /* errori             */
  double chi_2;
  char   file_name[41];            /* input file         */
  FILE  *in_file;

  printf("File con i dati: ");
  fgets(file_name, sizeof(file_name), stdin);
  file_name[strlen(file_name) - 1] = '\0';

  /* Apertura file di input */
  if ( (in_file = fopen(file_name,"r")) == NULL ) {
    fprintf(stderr, "\n%s : %s\n\n", file_name, strerror(errno));
    exit(errno);
  }

  /* inizializzazione variabili cumulate */
  sum_1  = 0;           /* sum 1  */
```

```
  sum_x  = 0.0;          /* sum x  */
  sum_y  = 0.0;          /* sum y  */
  sum_xx = 0.0;          /* sum xx */
  sum_xy = 0.0;          /* sum xy */
  sum_yy = 0.0;          /* sum yy */

  /* lettura coppie fino alla fine */
  while ( fscanf(in_file, "%lf%lf", &x, &y) == 2 ) {
    ++sum_1;
    sum_x  +=     x;
    sum_y  +=     y;
    sum_xx += x * x;
    sum_xy += x * y;
    sum_yy += y * y;
  }

  fclose(in_file);   /* il file non serve piu' */

  if (sum_1 == 1) {
    printf("\nNon abbastanza dati\n\n");
    exit(1);
  }

  /* Coefficienti retta a e b */
  det    =  sum_xx * sum_1 - sum_x * sum_x;
  coef_a = (sum_xx * sum_y - sum_x * sum_xy ) / det;
  coef_b = (sum_xy * sum_1 - sum_x * sum_y  ) / det;

  /* Errori sulla stima */
  if (sum_1 == 2) {
    sigma_a = 0.0;
    sigma_b = 0.0;
  }
  else {
    chi_2  =   sum_yy
             + coef_a * coef_a * sum_1
             + coef_b * coef_b * sum_xx;
    chi_2 -= 2.0 * (  coef_a * sum_y
                    + coef_b * sum_xy
                    - coef_a * coef_b * sum_x
                   );
    chi_2 /= (double) (sum_1 - 2);

    sigma_a = ( sum_xx / det )          * chi_2;
    sigma_b = ( (double) sum_1 / det )  * chi_2;
  }

  /* Output */
  printf("\nfit lineare: y = a + b x\n\n");
  printf("dati letti: %d\n", sum_1);
  printf("a         : %.3g +/- %.3g\n", coef_a, sqrt(sigma_a));
  printf("b         : %.3g +/- %.3g\n", coef_b, sqrt(sigma_b));
```

```
    printf("\n");

    return 0;
}
```

Note sul programma: `lin_fit.c`

- ```
  #include <stdlib.h>
  #include <math.h>
  #include <stdio.h>
  #include <string.h>
  #include <errno.h>
  ```

  Si includono i file di header necessari: `stdlib.h` per la funzione `exit()`, `math.h` per la funzione `sqrt()`, `stdio.h` per le funzioni di I/O `fopen()`, `fclose()` etc., `string.h` per le funzioni `strlen()` e`strerror()` e `errno.h` per la variabile di errore `errno`. Ricordiamo che la libreria matematica deve essere inclusa esplicitamente con il *flag* "`-lm`" del compilatore:

  ```
  $ cc lin_fit.c -lm
  ```

- ```
  char file_name[41];
  ```

  Per leggere il nome del file contenente le coppie di valori (x, y) si utilizza una stringa di 41 caratteri per cui il nome del file non può essere più lungo di 40 caratteri, un carattere è sempre utilizzato per il carattere '`\0`' di fine stringa. Se il nome eccede i 40 caratteri i caratteri in più non vengono letti.

- ```
  fgets(file_name, sizeof(file_name), stdin);
  file_name[strlen(file_name) - 1] = '\0';
  ```

  La funzione `fgets()` legge tutta la stringa di input incluso il carattere di *newline* alla fine che va quindi eliminato perchè non fa parte del nome del file. Dal momento che i nomi dei files sui sistemi UNIX sono composti solo da caratteri non bianchi è possibile utilizzare anche la funzione `scanf()` con la direttiva di conversione "`%s`", tuttavia l'uso della funzione `fgets()` mette a riparo dai noti problemi della funzione `scanf()` e permette di utilizzare il programma anche su sistemi che permettono l'uso di spazi bianchi nei nomi dei files.

- ```
  if ( (in_file = fopen(file_name,"r")) == NULL ) {
    fprintf(stderr, "\n%s : %s\n\n", file_name, strerror(errno));
    exit(errno);
  }
  ```

  Se vi è un errore in apertura del file questo viene segnalato e l'esecuzione del programma interrotta restituendo il valore della variabile `errno` che contiene il codice numerico dell'errore.

- ```
  sum_1  = 0;          /* sum 1  */
  sum_x  = 0.0;        /* sum x  */
  sum_y  = 0.0;        /* sum y  */
  sum_xx = 0.0;        /* sum xx */
  sum_xy = 0.0;        /* sum xy */
  sum_yy = 0.0;        /* sum yy */
  ```

Prima di iniziare il ciclo tutte le variabili su cui vanno cumulate le somme vengono azzerate in modo da non avere valori "spuri". È una buona forma di programmazione quella di mettere l'azzeramento delle variabili cumulate vicino al ciclo dove queste vengono utilizzate.

- ```
  while ( fscanf(in_file, "%lf%lf", &x, &y) == 2 ) {
  ```

Questa istruzione permette di leggere tutte le coppie di dati fino alla del file, o fino a quando una coppia non viene letta male, senza conoscerne a priori il loro numero. Ricordiamo che in lettura la direttiva di conversione per i dati di tipo `double` è "`%lf`". Se avessimo usato il controllo

```
while ( fscanf(in_file, "%lf%lf", &x, &y) == EOF ) {
```

questo ci avrebbe messo al riparo dalla fine del file ma non dalla possibilità che una coppia di dati possa essere letta male e possa pregiudicare così la stima dei coefficienti.

Al posto del ciclo `while` si sarebbe potuto utilizzare anche un ciclo `for` con la sola espressione di controllo:

```
for ( ; fscanf(in_file, "%lf%lf", &x, &y) == 2; ) {
```

È possibile includere l'incremento del contatore di coppie lette `sum_1` direttamente nel ciclo `for`:

```
for ( sum_1 = 0;
      fscanf(in_file, "%lf%lf", &x, &y) == 2; ++sum_1 ) {
```

In questo caso l'inizializzazione precedente di `sum_1` è superflua ma innocua mentre l'incremento di `sum_1` nel corpo del ciclo deve essere eliminato per non contare due volte le coppie.

- ```
  ++sum_1;
  sum_x  += x;
  sum_y  += y;
  sum_xx += x * x;
  sum_xy += x * y;
  sum_yy += y * y;
  ```

Gli ultimi valori letti vengono aggiunti alle somme calcolate con i valori letti precedentemente. In questo modo alla fine del ciclo le variabili cumulate contengono le somme valutate con tutti i dati letti.

- ```
  if (sum_1 == 1) {
    printf("\nNon abbastanza dati\n\n");
    exit(2);
  }
  ```

Per determinare i coefficienti a e b servono almeno due coppie di valori (x,y).

- ```
  if (sum_1 == 2) {
    sigma_a = 0.0;
    sigma_b = 0.0;
  }
  else {
    ...
  }
  ```

Se sono lette solo due coppie di dati l'errore è nullo, per due punti passa una ed una sola retta. Se vi sono più dati si calcola l'errore.

- ```
  printf("a          : %.3g +/- %.3g\n", coef_a, sqrt(sigma_a));
  printf("b          : %.3g +/- %.3g\n", coef_b, sqrt(sigma_b));
  ```

Per stampare i risultati usiamo il formato "%g" con al massimo 3 digits significativi dopo il punto decimale.

Test del programma: Per controllare i risultati del programma si utilizza un file con dei dati di cui si conosce il risultato, ad esempio un file con le coppie di dati generate con la regola $y = 2 + x$. In questo il risultato del programma deve essere

```
File con i dati: test.dat

fit lineare: y = a + b x

dati letti: 5
a         : 2 +/- 0
b         : 1 +/- 0
```

Esercizi

1. Il programma `lin_fit.c` produce un errore se il nome del file è più lungo di 40 caratteri. Modificare il controllo per determinare se è stato letto correttamente tutto il nome del file e non solo i primi 40 caratteri.

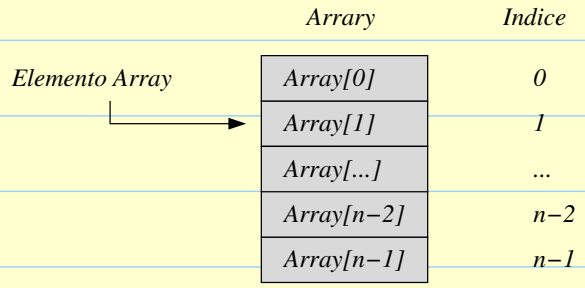
2.16. Array (Rev. 2.1.2)

Supponiamo dover leggere un certo numero di dati tutti dello stesso tipo da un file per poi manipolarli in qualche modo. Se i dati servono tutti insieme non è possibile utilizzare una o poche variabili per memorizzare a turno i dati e quindi è necessario definire una variabile per ogni dato che deve essere letto. Se il numero di dati è piccolo questa procedura non comporta troppi problemi, ma se il numero è grande la scrittura del programma diventa piuttosto faticosa. Inoltre, anche nel caso che il numero di dati sia piccolo, ogni cambiamento nel

numero di dati comporta la riscrittura del programma. È chiaro quindi che questa strategia non è la migliore.

In casi come questi, o più in generale ogni qual volta servano un certo numero di variabili tutte dello stesso tipo, è conveniente utilizzare il tipo `array`. Un `array` non è altro che una sequenza di locazioni di memoria consecutive in cui possono essere memorizzati oggetti tutti dello stesso tipo. Il numero di oggetti che possono essere memorizzati si chiama la dimensione dell'array, mentre ciascun oggetto viene chiamato `elemento` dell'array. La quantità di memoria occupata da un array è chiaramente data dalla dimensione dell'array moltiplicata per il numero di unità di memoria necessarie per contenere un elemento dell'array, ossia per la dimensione del tipo degli elementi dell'array.

Gli elementi di un array sono univocamente identificati da un numero chiamato indice che ne indica la loro posizione nell'array a partire dal primo elemento (*offset*), per cui se la dimensione dell'array è n il primo elemento avrà indice 0 e l'ultimo $n-1$ come mostrato nella figura seguente.



2.16.1. Dichiarazione del tipo array

La dichiarazione del tipo array di dimensione `size` di tipo `T` è della forma

```
type (array_name)[size]
```

dove `type` è il tipo `T` degli oggetti dell'array, `array_name` è l'identificatore o nome dell'array ed infine `size` è un valore intero positivo che specifica la dimensione dell'array, ossia il numero di elementi di tipo `T` che compongono l'array. Con l'esclusione di poche eccezioni, un array in C può essere formata di oggetti di qualsiasi tipo `T`.

Le parentesi "`()`" intorno all'identificatore non sono obbligatorie per cui in genere, a meno che non siano necessarie per la corretta interpretazione della dichiarazione, vengono omesse nella dichiarazione di un array. Ad esempio le due seguenti istruzioni

```
int data[3];
```

e

```
int (data)[3];
```

sono equivalenti ed entrambe dichiarano un array di dimensione 3 di tipo `int`, ossia composto da tre oggetti di tipo `int`, ed identificato dall'identificatore `data`.

La dichiarazione di un array riserva uno spazio di memoria per contenere gli `size` elementi dell'array e vi associa l'identificatore `array_name` per potervi accedere, ma non assegna un valore agli elementi che vanno quindi inizializzati prima di essere usati.[3]

Per poter riservare la memoria necessaria a contenere un array il compilatore deve conoscere quanti sono gli elementi dell'array, per cui nella dichiarazione di un array la dimensione `size` deve conosciuta ed espressa come valore numerico. Ad esempio la dichiarazione

```
int i = 3;
int data[i];
```

è illegale nello Standard C89. Nel C99 invece questa dichiarazione è permessa e gli arrays di questo tipo sono chiamati *variable length arrays*.

Per accedere ad un elemento di un array bisogna specificare il nome dell'array di appartenenza e l'indice dell'elemento tra parentesi quadre "`[]`":

$$\texttt{array\_name}[\texttt{index}] \qquad \texttt{index} = 0, \ldots, \texttt{size} - 1$$

per cui nel nostro esempio i tre elementi di `data` sono `data[0]`, `data[1]` e `data[2]`. Ricordiamo che nel linguaggio C, a differenza di altri linguaggi di programmazione, l'indice dell'array indica l'offset dell'elemento dell'array rispetto al primo elemento. Questo fatto è spesso causa di errori il cui effetto non è prevedibile a priori poiché in C non vi sono controlli sui valori degli indici degli arrays, come diventerà chiaro non appena verranno introdotti i puntatori.

Il seguente programma illustra l'uso degli arrays per calcolare la media di cinque numeri:

Programma: `array.c`

```
#include <stdio.h>

int main(void)
{
  int i;
  double data[5];          /* array di 5 elementi di tipo double */
  double     sum;          /* somma elementi array               */

  data[0] = 1.0;          /* assegnazione valori elementi array  */
  data[1] = 2.0;
  data[2] = 3.0;
  data[3] = 4.0;
  data[4] = 5.0;

  sum = 0.0;                    /* azzera la somma      */
  for(i = 0; i < 5; ++i) {
    sum += data[i];             /* somma tutti i valori */
  }
```

[3] Come nel caso delle variabili questo vale solo per gli array in classe di memorizzazione *automatica*, quelli in classe *statica* sono inizializzati automaticamente a zero.

```
    printf("Totale: %.3f  Media: %.3f\n", sum, sum / 5.0);
    return 0;
}
```

Per leggere i valore da terminale basta sostituire alle istruzioni che assegnano i valori agli elementi dell'array le seguenti istruzioni:

```
...
char line[81];

for(i = 0; i < 5; ++i) {
  printf("Dammi valore %d: ", i+1);
  fgets(line, sizeof(line), stdin);
  sscanf(line, "%lf", &data[i]);
}
...
```

Ricordiamo che per leggere dei dati di tipo `double` si deve utilizzare la direttiva di conversione "`lf`" e non semplicemente "`f`".

2.16.2. Inizializzazione di un array

Gli arrays possono essere inizializzati al momento della dichiarazione racchiudendo i valori da assegnare agli elementi dell'array tra parentesi graffe "{}"

```
type (array_name)[n] = { C_0, C_1, ..., C_(n-1) };
```

L'array viene inizializzata assegnando il valore di `C_i` all'elemento `i`-esimo `array_name[i]` dell'array. Nel precedente programma avremmo potuto quindi scrivere

```
double data[5] = {1.0, 2.0, 3.0, 4.0, 5.0};
```

Nello Standard C89 le `C_i` devono essere costanti di tipo `T'` il cui valore, eventualmente convertito con un cast implicito al tipo `T` dell'array, possa essere assegnato agli elementi dell'array. Nello Standard C99 è possibile utilizzare anche espressioni non costanti per le `C_i`.

Il numero di valori tra parentesi può essere inferiore della dimensione dell'array. In questo caso i valori forniti verranno assegnati agli elementi dell'array ad esaurimento, ai restanti elementi verrà assegnato un valore di *default* che per i tipi visti fino ad ora è il valore 0. Se invece il numero di valori è maggiore della dimensione dell'array è un errore che viene quindi segnalato dal compilatore con un messaggio, mentre la compilazione si interrompe o no a seconda del compilatore.

Se gli elementi dell'array vengono inizializzati contestualmente all dichiarazione dell'array la dimensione dell'array può essere omessa. Ad esempio, sempre nel programma precedente, avremmo potuto scrivere

```
double data[ ] = {1.0, 2.0, 3.0, 4.0, 5.0};
```

Un array di dimensione non specificata è un esempio di *tipo incompleto* poiché non tutte le informazioni sono state date. Il C permette di dichiarare variabili di questo tipo purché la

dichiarazione venga completata da una seconda dichiarazione che fornisca le informazioni mancanti. In questo caso è l'inizializzazione contestuale alla dichiarazione che fornisce l'informazione mancante sulla dimensione dell'array che viene presa uguale al numero di valori tra le parentesi graffe "{}". Di conseguenza l'istruzione precedente dichiara l'array `data` di `5` elementi di tipo `double`.

2.16.3. Arrays multidimensionali

Il linguaggio C permette di definire array con un numero qualsiasi di indici semplicemente aggiungendo dimensioni alla dichiarazione, purché ovviamente la memoria sia sufficiente. Di conseguenza la dichiarazione di un array *bidimensionale* di `size_1` $\times$ `size_2` elementi è

```
type (array_name)[size_1][size_2]
```

dove `type` è il tipo `T` degli elementi dell'array. Sintatticamente la precedente dichiarazione dichiara un array di dimensione `size_1` $\times$ `size_2` di tipo `T`. L'array ha due indici ed il primo prenderà i valori $0, \ldots,$ `size_1` -1 mentre il secondo i valori $0, \ldots,$ `size_2` -1. Ad esempio

```
double matrix[2][4];     /* una matrice di dimensione 2 x 4 */
```

definisce una matrice in due dimensioni 2×4. In modo analogo si dichiarano array con più indici per cui

```
double tensor[10][3][8][15];    /* un tensore quadridimensionale *
                                 * di dimensione 10 x 3 x 8 x 15  */
```

dichiara un array con quattro indici.

Gli elementi di un array multidimensionali sono identificati mediante il nome dell'array di appartenenza seguito dal valore degli indici racchiuso ciascuno dalle parentesi quadre "`[]`", per cui se ad esempio volessimo assegnare il valore 10 all'elemento $(2, 3)$ della matrice l'istruzione sarebbe:

```
matrix[1][2] = 10.00;
```

Ricordiamo che entrambi gli indici partono da 0 e non da 1 per cui l'elemento $(2, 3)$ della matrice è l'elemento dell'array con indici [1][2]. Osserviamo che contrariamente ad altri linguaggi di programmazione in C si usa la notazione `matrix[i][j]` e non `matrix[i,j]`. Anche questo diventerà chiaro una volta introdotti i puntatori.

Gli arrays in più dimensioni possono essere inizializzati contestualmente alla dichiarazione in un modo analogo a quello utilizzato per gli arrays con un solo indice con l'accortezza di incrementare gli indici da destra a sinistra e di raggruppare con parentesi graffe "{}" i valori da assegnare agli elementi con lo stesso gruppo di indici uguali, come mostrato dalla seguente istruzione

```
int matrix[2][3] = {
                      {1, 2, 3},
                      {4, 5, 6}
                    };
```

che dichiara l'array bidimensionale `matrix` di dimensione 2×3 ed assegna il valore 1 all'elemento `matrix[0][0]`, il valore 2 all'elemento `matrix[0][1]` e così via fino al valore 6 che viene assegnato a `matrix[1][2]`. I valori sono raggruppati in modo che il primo indice abbia sempre lo stesso valore. Nel caso di arrays con più indici la procedura è analoga solo si devono usare più parentesi, come mostrato nel caso di tre indici dalla seguente istruzione:

```
int tensor[2][2][3] = {
                        { {1, 2, 3}, {4, 5, 6} },
                        { {3, 2, 1}, {6, 5, 4} }
                      };
```

Chiaramente questo modo di assegnare i valori può diventare piuttosto pesante da scrivere se il numero di indici è elevato,

È possibile omettere completamente od in parte le parentesi per raggruppare i valori, ad esempio

```
int matrix[2][3] = {1, 2, 3, 4, 5, 6};  /* Warning in compilazione */
```

in questo caso generalmente il compilatore produce un messaggio di *warning*.

Se il numero di valori specificati è inferiore a quello degli elementi dell'array agli elementi in più viene assegnato il valore di default appropriato al tipo dell'array, che è il valore 0 per i tipi visti fino ad ora. Ad esempio l'istruzione

```
int matrix[2][3] = {
                      {1, 2, 0 },
                      {4, 0, 0 }
                    };
```

è perfettamente equivalente a

```
int matrix[2][3] = {
                      {1, 2},
                      {4}
                    };
```

A volte per inizializzare a zero tutti gli elementi di un array si trovano istruzioni del tipo:

```
double tensor[2][2][3][4] = {0};    /* Warning in compilazione */
```

In generale istruzioni di questo tipo producono un messaggio di *warning* in fase di compilazione. L'istruzione corretta è

```
double tensor[2][2][3][4] = {
                              { { {0}, {0}, {0} }, { {0}, {0}, {0} } },
                              { { {0}, {0}, {0} }, { {0}, {0}, {0} } }
                              };
```

Come nel caso degli arrays unidimensionali se una (o più) dimensione non viene specificata il suo valore viene dedotto, se possibile, dal numero di valori assegnati. Ad esempio la precedente istruzione poteva essere anche scritta come:

```
int tensor[ ][2][3] = {
                        { {1, 2, 3}, {4, 5, 6} },
```

```
                                    { {3, 2, 1}, {6, 5, 4} }
                                  };
```

Bisogna fare attenzione quando si omettono sia le dimensioni che alcuni valori da assegnare poiché il compilatore desume il valore delle dimensioni omesse a partire dal numero di valori assegnati per cui il risultato potrebbe essere diverso da quello voluto. Ad esempio la seguente istruzione

```
int matrix[ ][3] = { {1, 2} };    /* !!! Attenzione !!! */
```

dichiara una array bidimensionale di dimensione 1×3 ed assegna i valori 1, 2 e 0 ai suoi tre elementi. Se si vuole un array di dimensione 2×3 ed assegnare un valore non nullo solo ai primi due elementi l'istruzione corretta è

```
int matrix[ ][3] = { {1, 2}, {0} };    /* matrice 2 x 3 */
```

Il seguente programma

Programma: `array_bid.c`

```
#include <stdio.h>

int main(void)
{
  int array[3][2];          /* array bidimensionale di int */

  int i,j;

  array[0][0] = 0 * 10 + 0;
  array[0][1] = 0 * 10 + 1;

  array[1][0] = 1 * 10 + 0;
  array[1][1] = 1 * 10 + 1;

  array[2][0] = 2 * 10 + 0;
  array[2][1] = 2 * 10 + 1;

  printf("\n");
  for(i = 0; i < 3; ++i) {
    for(j = 0; j < 2; ++j) {
      printf("array[%d][%d] = %d\n", i, j, array[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

che stampa sullo schermo

```
array[0][0] = 0
array[0][1] = 1
```

```
array[1][0] = 10
array[1][1] = 11

array[2][0] = 20
array[2][1] = 21
```

illustra l'uso di arrays multidimensionali.

Qualificatori `const` e `volatile`

Nella dichiarazione di un array è possibile utilizzare sia il qualificatore `const` che il qualificatore `volatile`. Questi si applicano separatamente ad ogni elemento dell'array, per cui ad esempio

```
const int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

dichiara un array bidimensionale i cui elementi sono `const int`. Questo vuol dire che il valore di ciascun elemento non può essere cambiato, per cui un'eventuale istruzione

```
matrix[1][1] = 3;
```

produce un messaggio di errore in fase di compilazione.

2.17. Stringhe (Rev. 2.1.1)

Una stringa è una generica sequenza di di caratteri. Più specificatamente in nel linguaggio C per stringa si intende sequenza di zero o più caratteri terminata dal carattere nullo '`\0`' (`NUL`) che indica la fine della stringa. Abbiamo già incontrato le stringhe come costanti stringa, ma il loro uso non è limitato alle costanti.

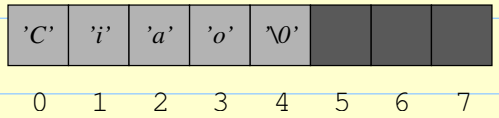
2.17.1. Dichiarazione

Il linguaggio C non introduce un tipo particolare per le stringhe ma le rappresenta utilizzando arrays di tipo `char`. Di conseguenza la dichiarazione di una variabile di tipo stringa o semplicemente stringa è della forma:

```
char string_name[size]
```

dove `string_name` è l'identificatore della stringa e `size` è il numero di caratteri della stringa. Questa dichiarazione necessita però di una precisazione. Infatti a ben guardare questa è la dichiarazione di un array unidimensionale di `size` elementi di tipo `char` mentre una stringa è una sequenza di caratteri *terminata* dal carattere nullo '\0'. Strettamente parlando la definizione precedente definisce la variabile `string_name` che può contenere stringhe di lunghezza massima `size-1`, ossia stringhe composte da 0 fino ad un massimo di `size-1` caratteri.

La seguente figura mostra come la stringa `"Ciao"` viene rappresentata con un array di tipo `char` di dimensione 8:



Gli ultimi tre elementi dell'array, di indice 5, 6 e 7, non sono utilizzati e possono contenere qualsiasi cosa. Questo non crea problemi nella rappresentazione della stringa perché in ogni caso la lettura della stringa di ferma quando viene letto il carattere '\0' nel quinto elemento dell'array, indice 4.

Per contro per rappresentare una stringa di lunghezza `length` è necessario un array di tipo `char` di dimensione minima `length+1`. Ad esempio per rappresentare la stringa di 4 caratteri `"Ciao"` serve un array di almeno 5 elementi di tipo `char`: 4 per i caratteri '`C`', '`i`', '`a`', '`o`', ed 1 per il carattere nullo '\0'.

2.17.2. Inizializzazione

Dal momento che le stringhe sono rappresentate come arrays di tipo `char` queste possono essere inizializzate come un qualsiasi array assegnando a ciascun elemento un carattere della stringa, ed aggiungendo il carattere nullo '\0' alla fine. Ad esempio per assegnare il valore `"Ciao"` ad una stringa si può procedere come:

```
char string[5];      /* Array dim. 5 -> Stringa lung. 4 */

string[0] = 'C';
string[1] = 'i';
string[2] = 'a';
string[3] = 'o';
string[4] = '\0';
```

Alternativamente è anche possibile utilizzare l'istruzione

```
char string[5] = {'C', 'i', 'a', 'o', '\0'};
```

ovvero

```
char string[ ] = {'C', 'i', 'a', 'o', '\0'};
```

In entrambi i casi viene dichiarata una variabile di tipo stringa `string` di 4 caratteri a cui viene assegnato il valore `"Ciao"`.

Qualsiasi di queste forme si usi il carattere nullo '\0' deve essere sempre aggiunto alla fine altrimenti `string` verrebbe interpretato come un array di caratteri e non come una stringa.

Va da se che questo modo di assegnare il valore alle stringhe non è dei più agevoli, ad esempio è facile dimenticarsi una virgola o un apice se non il carattere nullo. Per semplificare le cose il linguaggio C permette la dichiarazione ed inizializzazione delle stringhe nelle seguenti forme equivalenti, ma ben più comode,

```
char string[5] = "Ciao";
char string[ ] = "Ciao";
```

Nella dichiarazione di una variabile di tipo stringa non è necessario che la dimensione dell'array di tipo `char` sia esattamente uguale a quella della stringa da memorizzare, più uno per il carattere nullo '\0'. L'unica richiesta è che la dimensione sia sufficientemente grande per contenere la stringa, per cui l'istruzione

```
char string[80] = "Ciao";
```

è perfettamente lecita. In questo caso `string` può contenere fino ad 80 caratteri, incluso il carattere nullo alla fine, anche se per la stringa `"Ciao"` ne usano solo 5, gli altri sono semplicemente inutilizzati.

Questo modo di assegnare un valore alla stringa è permesso solo all'atto della dichiarazione, infatti un'istruzione del tipo

```
char string[5];

string = "Ciao";    /* Illegale !!! */
```

è illegale poiché `string` è pur sempre un array. Per lo stesso motivo anche l'istruzione seguente per assegnare il valore di una stringa ad un'altra

```
char str_1[5] = "Ciao";
char str_2[5];

str_2 = str_1;    /* Illegale !!! */
```

non è permessa.

Se si vuole assegnare un valore ad una stringa dopo che questa è stata dichiarata, o cambiarne il valore, si deve procedere come per un array qualsiasi assegnando ad ogni elemento della stringa il carattere corrispondente. Così ad esempio la forma corretta dell'assegnazione precedente è

```
int i;
char str_1[5] = "Ciao";
char str_2[5];

for (i = 0; i < 5; ++i)
  str_2[i] = str_1[i];    /* Legale !!! */
```

Chiaramente utilizzare le stringhe in questo modo può essere piuttosto noioso. Per ovviare a ciò le librerie standard forniscono tutta una serie di funzioni che permettono di operare con le stringhe in modo piuttosto semplice. Queste funzioni sono usualmente definite nel file di header `string.h`. Noi ne considereremo solo alcune.

Funzione strcpy()

Per assegnare un valore ad una stringa si può utilizzare la funzione

```
#include <string.h>

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

che copia il contenuto della stringa `src` nella stringa `dest`. La stringa `src` rimane invariata mentre il valore precedente della stringa `dest` viene perso. Tutti i caratteri della stringa `src`, incluso il carattere nullo '`\0`, sono copiati anche se la stringa `src` è più lunga della stringa `dest` per cui è cura del programmatore assicurarsi che la stringa `dest` sia di lunghezza sufficiente. La funzione `strcpy()` ritorna sempre l'indirizzo di memoria (puntatore) del primo elemento dell'array `dest`.

Utilizzando la funzione `strcpy()` l'esempio precedente può essere riscritto in forma più chiara e concisa come

```
char str_1[10];
char str_2[5];

strcpy(str_1, "Ciao");
strcpy(str_2, str_1);
```

In questo esempio la dimensione minima di `str_2` è 5 perché `str_1` contiene 4 caratteri più il carattere nullo '`\0`'.

Per assegnare un valore ad una stringa si può utilizzare anche la funzione `sprintf()`, ad esempio al posto delle precedenti istruzioni si sarebbe potuto utilizzare

```
char str_1[10];
char str_2[5];

sprintf(str_1, "%s", "Ciao");
sprintf(str_2, "%s", str_1);
```

A questo livello le due funzioni sono sostanzialmente equivalenti. La funzione `sprintf()` è tuttavia più flessibile perché permette di assegnare ad una stringa valori ottenuti a partire da variabili anche di tipo diverso da tipo stringa, ad esempio di tipo `int`, come mostra il seguente semplice programma

Programma: `sprintf.c`

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

int main(void)
{
  char str[10];
  int        i;

  for (i = 1; i <= 4; ++i) {
    sprintf(str, "Ciao-%02d", i);
    printf("str -> \"%s\"\n", str);
  }

  return 0;
}
```

Quando il programma viene eseguito scrive sul terminale la stringa ottenuta a partire dalla stringa "Ciao-" e dal valore della variabile i di tipo int scritto come due digits decimali con allineamento a destra e 0-padding ("%02d"):

```
str -> "Ciao-01"
str -> "Ciao-02"
str -> "Ciao-03"
str -> "Ciao-04"
```

Questa possibilità si rivela particolarmente utile quando il nome di uno stream, ad esempio un file, può variare a seconda del valore di una o più variabili.

Funzione strcat()

Due stringhe possono essere "concatenate" utilizzando la funzione

```
#include <string.h>

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

che appende il contenuto della stringa str alla fine della stringa dest. Il carattere nullo alla fine della stringa dest, come pure il contenuto degli elementi successivi dell'array utilizzata per rappresentare la stringa, sono sovrascritti con il contenuto della stringa str. Tutti i caratteri della stringa str, incluso il carattere nullo '\0' sono copiati di seguito a quelli della stringa dest che quindi deve essere di dimensione sufficiente per poter contenere le due stringhe insieme. La stringa src rimane invariata. La funzione strcat() ritorna sempre l'indirizzo di memoria (puntatore) del primo elemento dell'array dest.

Il seguente programma mostra l'uso della funzione strcat().

Programma: strcat_1.c

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

int main(void)
{
  char str_1[50];
  char str_2[50];

  strcpy(str_1, "Prima stringa");
  strcpy(str_2, "Seconda stringa");

  printf("str_1 : \"%s\"\n", str_1);
  printf("str_2 : \"%s\"\n", str_2);

  strcat(str_1, str_2);
  printf("str_1+str_2 : \"%s\"\n", str_1);

  return 0;
}
```

Quando il programma viene eseguito si ha

```
str_1 : "Prima␣stringa"
str_2 : "Seconda␣stringa"
str_1+str_2 : "Prima␣stringaSeconda␣stringa"
```

Osserviamo che la seconda stringa viene copiata di seguito alla prima senza inserire nessun separatore tra le due. Se le stringhe vanno separate questo deve essere fatto esplicitamente come mostra il seguente programma

Programma: `strcat_2.c`

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

int main(void)
{
  char str_1[50];
  char str_2[50];

  strcpy(str_1, "Prima stringa");
  strcpy(str_2, "Seconda stringa");

  printf("str_1 : \"%s\"\n", str_1);
  printf("str_2 : \"%s\"\n", str_2);

  strcat(str_1, " ");          /* aggiunge uno spazio fine str_1 */
  strcat(str_1, str_2);
  printf("str_1+str_2 : \"%s\"\n", str_1);

  return 0;
}
```

Quando il programma viene eseguito si ha

```
str_1 : "Prima␣stringa"
str_2 : "Seconda␣stringa"
str_1+str_2 : "Prima␣stringa␣Seconda␣stringa"
```

Gli spazi bianchi sono stati evidenziati per chiarezza.

Funzione strcmp()

Per conoscere se due stringhe sono uguali si può utilizzare la funzione

```c
#include <string.h>

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

che effettua un confronto lessicografico del contenuto della stringa `s1` con quello della stringa `s2`. Il confronto lessicografico viene effettuato confrontando il valore di ciascun carattere

secondo la codifica utilizzata, così ad esempio nel codice ASCII la stringa `"A"` è minore della stringa `"a"` poiché il valore decimale del carattere '`A`' è 65 mentre quello del carattere '`a`' è 97.

La funzione `strcmp()` restituisce un valore di tipo `int` che è negativo se `s1` è minore di `s2`, positivo se `s1` è maggiore di `s2` e zero se `s1` è uguale a `s2`. Due stringhe sono uguali se sono composte da esattamente gli stessi caratteri.

Le seguenti istruzioni mostrano come effettuare il confronto tra due stringhe:

```
if ( strcmp(str_1, str_2) == 0 )
   printf("Stringhe uguali\n");
else
   printf("Stringhe diverse\n");
```

Spesso sfruttando il fatto che ogni valore diverso da zero viene interpretato come un'affermazione vera si utilizza la forma equivalente

```
if ( !strcmp(str_1, str_2) )
   printf("Stringhe uguali\n");
else
   printf("Stringhe diverse\n");
```

anche se meno chiara.

Funzione strlen()

La funzione

```
#include <string.h>

size_t strlen(const char *str);
```

calcola la lunghezza della stringa `str`. La funzione `strlen()` ritorna un intero non negativo uguale al numero di caratteri della stringa `str` che precedono il carattere nullo '`\0`'. Una stringa vuota è composta dal solo carattere nullo per cui la sua lunghezza è zero.

Non bisogna confondere la lunghezza di una stringa con la sua dimensione. La lunghezza di una stringa è il numero di caratteri che la compongono, escluso il carattere nullo '`\0`', mentre la dimensione della stringa è il numero di elementi dell'array di tipo `char` utilizzato per rappresentarla. Il seguente programma illustra questa differenza utilizzando la funzione `strlen()` per determinare la lunghezza della stringa e l'operatore `sizeof` per determinarne la dimensione, ossia il numero di elementi dell'array di tipo `char` utilizzato per rappresentarla.

Programma: `strcpy_len.c`

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

int main(void)
{
  char string[50];
```

```
  int      length;

  strcpy(string, "Ciao");
  length = strlen(string);

  printf("La lunghezza della stringa \"%s\" e' %d\n", string, length);
  printf("La dimensione della stringa string e' %d\n", sizeof(string));

  return 0;
}
```

Quando il programma viene eseguito si ha

```
La lunghezza della stringa "Ciao" e' 4
La dimensione della stringa string e' 50
```

La stringa `string` è infatti composta da 4 caratteri anche se viene utilizzato un array di 50 caratteri per rappresentarla.

2.17.3. Input/Output

Tutte le operazioni di Input/Output con stringhe e caratteri si possono effettuare utilizzando le funzioni della famiglia `scanf()` e `printf()` con le opportune direttive di conversione: "`%c`" e "`%s`". Entrambe le direttive posso includere dei caratteri opzionali per specificare la lunghezza del campo, l'allineamento, in numero di caratteri da leggere e così via. Rimandiamo alla parte sulle operazioni di Input/Output per una discussione più dettagliata di queste direttive di conversione. Il seguente programma mostra qualche esempio di formattazione di stampa per caratteri e stringhe.

Programma: `print_char.c`

```
#include <stdio.h>

int main(void)
{
  char car   = 'A';
  char str[] = "Hello World!";


  printf("char   format %%c      : \"%c\"\n", car);
  printf("char   format %%5c     : \"%5c\"\n", car);
  printf("char   format %%-5c    : \"%-5c\"\n", car);
  printf("string format %%s      : \"%s\"\n", str);
  printf("string format %%3s     : \"%3s\"\n", str);
  printf("string format %%.7s    : \"%.7s\"\n", str);
  printf("string format %%10.7s  : \"%10.7s\"\n", str);
  printf("string format %%-10.7s : \"%-10.7s\"\n", str);

  return 0;
}
```

Quando questo programma viene eseguito produce il seguente output:

```
char   format %c       : "A"
char   format %5c      : "    A"
char   format %-5c     : "A    "
string format %s       : "Hello World!"
string format %3s      : "Hello World!"
string format %.7s     : "Hello W"
string format %10.7s   : "   Hello W"
string format %-10.7s  : "Hello W   "
```

infatti

Conversione	Variabile	Output	Note
%c	car	"A"	Dimensione del campo 1, campo minimo richiesto
%5c	car	"␣␣␣␣A"	Dimensione del campo 5, allineato a destra
%-5c	car	"A␣␣␣␣"	Dimensione del campo 5, allineato a sinistra, padding *spazio*
%s	str	"Hello␣World!"	Dimensione del campo 12, campo minimo richiesto
%3s	str	"Hello␣World!"	Dimensione del campo 3, non sufficiente aumentato al minimo richiesto
%.7s	str	"Hello␣W"	Precisione 7, campo 7, minimo richiesto
%10.7s	str	"␣␣␣Hello␣W"	Dimensione campo 10, precisione 7, allineato a destra
%-10.7s	str	"Hello␣W␣␣␣"	Dimensione campo 10, precisione 7, allineato a sinistra

In questo esempio i doppi apici sono stati messi solo per evidenziare la dimensione del campo.

Esistono tuttavia alcune funzioni specifiche per le operazioni di Input/Output con stringhe e caratteri.

Funzioni: fputc() e fputs()

La funzione

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

scrive sullo stream identificato da `stream` il carattere `c`. Se non vi sono errori la funzione restituisce il valore del carattere scritto come valore di tipo `int` altrimenti restituisce `EOF`.

Ogni chiamata della funzione `fputc()` avanza di una posizione il puntatore che indica la posizione di scrittura lungo lo stream di conseguenza chiamate successive della funzione scrivono i caratteri i posizioni successive.

La funzione

```
#include <stdio.h>

int fputs(const char *str, FILE *stream);
```

scrive sullo stream identificato da `stream` tutti i caratteri della stringa `str` fino al carattere nullo escluso. Se non vi sono errori la funzione restituisce il valore nullo, o un altro valore non negativo a seconda del sistema, altrimenti restituisce il valore `EOF`. Su alcuni sistemo se la stringa `str` è vuota la funzione `fputs()` restituisce un valore indeterminato.

Ogni chiamata della funzione `fputs()` avanza il puntatore che indica la posizione di scrittura lungo lo stream alla prima posizione subito dopo l'ultimo carattere scritto cosicché chiamate successive della funzione scrivono le stringhe una di seguito all'altra.

Per scrivere una stringa sullo stream standard di output `stdout` può essere più comodo utilizzare la funzione

```
#include <stdio.h>

int puts(const char *str);
```

La funzione `puts()` differisce dalla funzione `fputs()` per il fatto che l'output è sempre sullo stream `stdout` e che dopo ogni stringa viene sempre scritto il carattere di newline cosicché chiamate successive della funzione scrivono le stringhe su righe successive.

Funzioni: fgetc() e fgets()

Per leggere un solo carattere da uno stream di input si può usare la funzione

```
#include <stdio.h>

int fgetc(FILE *stream);
```

che legge un carattere dallo stream `stream` e lo restituisce convertito in `int`. Se se verifica un errore, oppure si è raggiunta la fine del file la funzione ritorna `EOF`. Ogni chiamata della funzione sposta di una posizione il puntatore che indica la posizione lungo lo stream per cui chiamate successive della funzione `fgetc()` leggono i caratteri dallo stream in successione.

Il seguente programma illustra l'uso delle funzioni `fgetc()` e `fputc()` scrivendo due volte (*echo*) quanto scritto sul terminale:

Programma: `echo.c`

```
#include <stdio.h>

int main(void)
{
```

```
  int c;

  while( (c = fgetc(stdin)) != EOF) {
    if (c == '\n') break;
    fputc(c, stdout);
    fputc(c, stdout);
  }
  printf("\n");
  return 0;
}
```

Per leggere una stringa si utilizza la funzione

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
```

già incontrata nella discussione sull'Input/Output. La funzione `fgets()` legge al massimo `size-1` caratteri dallo stream associato all'identificatore `stream` e li scrive sulla stringa individuata dall'identificatore `s`. La lettura si ferma quando viene letto un `EOF` o un *newline* ovvero se sono stati letti `size-1` caratteri. In ogni caso dopo l'ultimo carattere letto viene sempre aggiunto il carattere nullo '`\0`'.

Se la lettura dallo stream termina senza problemi la funzione `fgets()` restituisce come valore l'indirizzo di memoria della stringa `str` (*puntatore*). Se non viene letto nulla la funzione restituisce il valore `NULL` (puntatore nullo) ed il contenuto della stringa `str` non viene modificato. Il valore `NULL` viene restituito anche nel caso di errore nell'operazione di input ma in questo caso il contenuto della stringa `str` non è prevedibile. La funzione `fgets()` non distingue tra un errore di input e la fine dello stream (*end-of-file*).

Se la lettura termina perché è stato letto un *newline* il carattere di *newline* viene scritto come ultimo carattere della stringa, subito prima del carattere nullo, come mostrato dal seguente programma.

Programma: `fgets_wrong.c`

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

int main(void)
{
  char line[100];

  printf("Scrivi una linea: ");
  fgets(line, sizeof(line), stdin);

  printf("\nHai scritto: \"%s\"\n", line);
  printf("ed e' lunga %d bytes\n", strlen(line));

  return 0;
```

```
}
```

Quando il programma viene eseguito si ha il seguente risultato

```
Scrivi una linea: Ciao

Hai scritto: "Ciao
"
ed e' lunga 5 bytes
```

Come mai il carattere "" è finito su una nuova linea? Il motivo è che la funzione `fgets()` legge tutta la stringa incluso il newline 'NL' alla fine della linea di input.[4] Questo vuol dire alla stringa `line` è stato assegnato il valore:

```
line[0] = 'C';
line[1] = 'i';
line[2] = 'a';
line[3] = 'o';
line[4] = '\n';
line[5] = '\0';
```

come anche segnalato dalla sua lunghezza che è 5 e non 4. Di conseguenza se vogliamo scrivere correttamente la stringa letta dobbiamo eliminare il newline. Siccome il newline è l'ultimo carattere della stringa questo viene eliminato facilmente con l'istruzione:

```
line[strlen(line)-1] = '\0';
```

che sposta la fine della stringa di un carattere a sinistra. La versione corretta del programma è quindi

Programma: `fgets_correct.c`

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

int main(void)
{
  char line[100];

  printf("Scrivi una linea: ");
  fgets(line, sizeof(line), stdin);

  line[strlen(line)-1] = '\0';

  printf("\nHai scritto: \"%s\"\n", line);
  printf("ed e' lunga %d bytes\n", strlen(line));

  return 0;
}
```

[4] Su sistemi MS-DOS/Windows il newline è composto dai due caratteri CR e NL.

Quando il programma viene eseguito adesso si ha correttamente

```
Scrivi una linea: Ciao

Hai scritto: "Ciao"
ed e' lunga 4 bytes
```

È possibile leggere una stringa anche utilizzando le funzioni della famiglia `scanf` tuttavia in questo caso se non si fa attenzione il risultato potrebbe essere diverso da quello voluto. Il motivo è che la direttiva di conversione "`%s`" legge una sequenza di caratteri *non bianchi* per cui se la stringa da leggere è composta da due o più parole separate da spazi bianchi solo la prima viene letta, come mostra il seguente programma.

Programma: `fgets_vs_scanf.c`

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

int main(void)
{
  char line[100];

  printf("Scrivi una linea :");
  fgets(line, sizeof(line), stdin);

  line[strlen(line)-1] = '\0';
  printf("fgets ha letto  :\"%s\"\n", line);

  printf("\n(Ri)Scrivi la linea :");
  fscanf(stdin,"%s", line);

  printf("scanf ha letto     :\"%s\"\n", line);
  return 0;
}
```

Quando il programma viene eseguito si ha

```
Scrivi una linea :Ciao a tutti
fgets ha letto  :"Ciao a tutti"

(Ri)Scrivi la linea :Ciao a tutti
scanf ha letto     :"Ciao"
```

Un'altra caratteristica della direttiva di conversione "`%s`" è quella di ignorare gli spazi bianchi all'inizio della stringa. Se infatti al nostro programma viene data la stringa "␣␣Ciao" si ha

```
Scrivi una linea :  Ciao
fgets ha letto  :"␣␣Ciao"

(Ri)Scrivi la linea :  Ciao
scanf ha letto     :"Ciao"
```

Gli spazi bianchi sono stati evidenziati per una maggiore chiarezza.

La direttiva di conversione "%c" non soffre di questi problemi però è bene ricordarsi che questa direttiva anche quando è utilizzata per leggere più di un carattere non aggiunge il carattere nullo '\0' alla fine della stringa. Il seguente programma mostra questo comportamento

Programma: `scanf_c.c`

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

int main(void)
{
  int       i;
  char str[10];

  printf("Scrivi una stringa  :");
  scanf("%4c", str);
  printf("la stringa contiene :\n");
  for (i = 0; i < 10; ++i)
    printf("                      '%c' -> %4d (%#x)\n",
           str[i], str[i], str[i]);

  return 0;
}
```

Quando il programma viene eseguito fornendo da tastiera la stringa "␣␣Ciao" si ha

```
Scrivi una stringa  :  Ciao
la stringa contiene :
                      ' ' ->   32 (0x20)
                      ' ' ->   32 (0x20)
                      'C' ->   67 (0x43)
                      'i' ->  105 (0x69)
                      ' ' ->  -64 (0xffffffc0)
                      ' ' ->  -24 (0xffffffe8)
                      ' ' ->  -65 (0xffffffbf)
                      ' ' ->  -65 (0xffffffbf)
                      '' -> -128 (0xffffff80)
                      ' ' ->  -24 (0xffffffe8)
```

Da questo output, che mostra per ciascun carattere il suo codice sia in formato decimale che esadecimale, si vede facilmente che sono stati letti i primi 4 caratteri della stringa senza saltare gli spazi bianchi: due *space* (codice ASCII esadecimale 20), il carattere 'C' (codice ASCII esadecimale 43) ed il carattere 'i' (codice ASCII esadecimale 69). Inoltre dopo i caratteri letti non vi è il carattere nullo '\0' (codice ASCII 0). Il contenuto degli elementi che seguono i quattro caratteri letti non è prevedibile. Nell'esempio contiene valori corrispondenti a caratteri non stampabili.

Se al posto della direttiva di conversione "%4c" fosse stata usata la direttiva "%s" il risultato sarebbe stato

```
Scrivi una stringa  :  Ciao
la stringa contiene :
                      'C' ->   67 (0x43)
                      'i' ->  105 (0x69)
                      'a' ->   97 (0x61)
                      'o' ->  111 (0x6f)
                      '' ->    0 (0)
                      ' ' ->  -24 (0xffffffe8)
                      ' ' ->  -65 (0xffffffbf)
                      ' ' ->  -65 (0xffffffbf)
                      '' -> -128 (0xffffff80)
                      ' ' ->  -24 (0xffffffe8)
```

perché gli spazi bianchi iniziali sono ignorati, ed inoltre si riconosce facilmente il carattere nullo '\0'. Di nuovo gli elementi inutilizzati contengono valori non prevedibili.

2.18. Il Preprocessore C (Rev. 2.1.2)

Il preprocessore C nasce dall'esigenza di poter modificare in modo semplice ed efficace una o più parti di un programma per poterlo adattare ad esigenze differenti. Il seguente esempio mostra una situazione tipica di utilizzo del preprocessore. Consideriamo il programma

```
int main(void)
{
  int          i;
  int index[10];

  for (i = 0; i < 10; ++i) {
    index[i] = 2 * i;
  }

  return 0;
}
```

in cui viene definito l'array `index` di tipo `int` di dimensione 10. Se adesso fosse necessario dover cambiare la dimensione dell'array `index`, diciamo a 25, bisognerebbe modificare il programma e sostituire al valore 10 il valore 25 sia nella dichiarazione dell'array che nel ciclo `for`. In questo caso il programma è molto semplice e si tratta di effettuare due sole sostituzioni, ma è facile rendersi conto che la procedura può diventare non solo laboriosa ma anche ad alta probabilità di errore per programmi più complessi in cui il numero di sostituzioni da effettuare può essere elevato e trovarsi in varie parti del programma molto distanti tra loro o su files sorgente differenti.

In questo caso si potrebbe risolvere il problema utilizzando una scrittura del tipo

```
int main(void)
{
  const int SIZE = 10;
  int           i;
  int index[SIZE];
```

```
    for (i = 0; i < SIZE; ++i) {
      index[i] = 2 * i;
    }

    return 0;
}
```

in cui si utilizza un array di dimensione variabile. Sfortunatamente lo Standard C89 non permette di dichiarare arrays di dimensione variabile, anche nel caso in cui la variabile che contiene le dimensioni sia dichiarata costante aggiungendo il qualificatore `const`. Gli arrays di dimensione variabile sono stati introdotti nel linguaggio C dallo Standard C99. Un modo di risolvere il problema senza utilizzare arrays di dimensione variabile, e quindi restando nello Standard C89, è come vedremo quello di ricorrere al preprocessore C. L'utilità del preprocessore C non è tuttavia limitata a casi di questo tipo ma è molto più ampia.

Il preprocessore C è sostanzialmente un editore di testo che legge il file sorgente prima del compilatore C vero e proprio e lo modifica secondo le direttive contenute (*embedded*) nel file stesso producendo un nuovo file sorgente con solo istruzioni in linguaggio C. In alcuni sistemi il preprocessore è un programma separato dal compilatore che legge il file sorgente originale e produce il nuovo file sorgente "preprocessato" che viene poi usato come input per il compilatore C. Su altri sistemi invece un solo programma effettua tutte le operazioni ossia legge, modifica e compila il file sorgente senza la creazione di un file sorgente preprocessato "intermedio".

2.18.1. Comandi e direttive

L'operato del preprocessore C viene controllato scrivendo le direttive direttamente nel file, o files, sorgente del programma. Questo non crea confusione perché le direttive sono interpretate dal preprocessore C che le elimina e produce il file sorgente preprocessato con solo istruzioni C, per cui il compilatore C vero e proprio non vedrà mai le direttive del preprocessore.

Le direttive del preprocessore vengono indicate nel file sorgente dal il carattere *hash* "`#`" posto all'inizio della linea: ogni linea del file sorgente che inizia con il carattere "`#`" viene interpretata come linea di comando o direttiva per il preprocessore C.

Il carattere "`#`" all'inizio della linea deve essere seguito dal comando del preprocessore. Lo standard C permette di inserire spazi bianchi sia prima che dopo il carattere "`#`", tuttavia alcuni compilatori più vecchi richiedono che il carattere "`#`" sia sempre all'inizio della riga, ossia in prima colonna, e che non vi siano spazi bianchi tra "`#`" ed il nome del comando.

La sintassi dei comandi del preprocessore C è completamente indipendente, anche se simile, da quella del linguaggio C ed inoltre la struttura delle direttive è in genere più rigida del C. Se il comando prende un argomento la parte della linea che segue il comando fino al *newline* viene interpretata come argomento del comando. Se invece il comando non prende argomenti la linea deve contenere dopo il comando solo caratteri bianchi fino al *newline*. Eventuali commenti presenti sulla linea, anche se estesi su più linee, sono sostituiti con un singolo spazio bianco per cui non vengono considerati come parte della direttiva del preprocessore C. Ad esempio le direttive

```
#include <stdio.h>    /* Commento */
#include <math.h>     /* Altro
                         Commento */
#include /* commento */ <stdlib.h>
```

sono interpretate come direttive

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

Una direttiva del preprocessore può essere scritta su più linee ponendo immediatamente prima del *newline* che termina la linea il *backslash* "\". In questo caso la linea che segue, anche se inizia con il carattere "#", viene interpretata come parte della direttiva della linea precedente. Ad esempio se il *backslah* "\" nelle seguenti linee

```
#define BACKSLASH \
#define END       }
```

precede immediatamente il *newline* il secondo comando `define` viene ignorato e le due linee vengono interpretate come un'unica direttiva del preprocessore:

```
#define BACKSLASH #define END       }
```

Le direttive sono interpretate dal preprocessore C sequenzialmente nell'ordine con cui sono scritte nel file sorgente per cui in un qualsiasi punto del file è noto solo il risultato delle direttive già interpretate non di quelle che seguono.

Infine è bene ricordare che il preprocessore C non riconosce e quindi non controlla la sintassi del linguaggio C. Questo è spesso causa di errori di difficile individuazione. In questi casi è utile poter vedere il risultato del preprocessore C sul file, questo si ottiene facilmente utilizzando il *flag* "`-E`" del compilatore:

```
$ cc -E program_file.c
```

in modo che il file preprocessato non venga inviato al compilatore ma sul terminale.

Il preprocessore C meriterebbe una trattazione a parte ma questo ci porterebbe troppo lontano dai nostri scopi, per cui in questo primer ci limiteremo a considerare solo alcuni dei comandi del preprocessore trascurando completamente ad esempio il capitolo delle macros di sistema.

I comandi che considereremo sono:

Comando	**Azione**
`#define`	Definisce una macro
`#undef`	Cancella la definizione di una macro
`#include`	Inserisce delle linee di testo da un altro file
`#if`	Inserisce delle linee di testo a seconda del valore di un espressione costante
`#ifdef`	Inserisce delle linee di testo se il nome di una macro è definito

Comando	Azione
`#ifndef`	Inserisce delle linee di testo se il nome di una macro non è definito
`#else`	Inserisce delle linee di testo se il precedente `#if`, `#ifdef`, `#ifndef` o `#elif` è risultato falso
`#elif`	Inserisce delle linee di testo a seconda del valore di espressione costante se il precedente `#if`, `#ifdef`, `#ifndef` o `#elif` è risultato falso
`#endif`	Termina un inserimento di testo condizionato

2.18.2. Comando `#define`

Il comando `#define` definisce una *macro* del preprocessore associando ad un identificatore, il nome della macro, una sequenza di caratteri (*string-of-text*) chiamato *corpo* della macro. Il corpo può anche essere vuoto, ossia composto da nessun carattere. Quando il file viene letto dal preprocessore C ogni occorrenza della macro nel file, esclusa ovviamente la sua definizione, viene *espansa* sostituendo al nome della macro il corpo.

Le macros possono essere definite sia senza che con parametri.

Macros senza parametri

La sintassi del comando `#define` per definire una macro senza parametri è

```
#define macro_name string-of-text  /* comment */
```

dove `macro_name` è il nome della macro e `string-of-text` il corpo. Siccome il preprocessore C, al pari del compilatore C, tratta i commenti come un singolo spazio bianco, il commento `comment` non viene considerato come parte del corpo della macro. Il preprocessore sostituirà nel resto del file, ossia dalla definizione in poi, ogni occorrenza di `macro_name` con `string-of-text`. Ad esempio la seguente linea di comando per il preprocessore

```
#define SIZE 10
```

definisce la macro `SIZE`. Di conseguenza il preprocessore ogni qual volta troverà nel resto del programma "`SIZE`" vi sostituirà "`10`". Possiamo quindi scrivere una versione più flessibile del precedente esempio come:

```
#define SIZE 10
int main(void)
{
  int           i;
  int index[SIZE];

  for (i = 0; i < SIZE; ++i) {
    index[i] = 2 * i;
  }
```

```
  return 0;
}
```

per cui adesso per cambiare la dimensione dell'array in modo consistente all'interno del file è sufficiente modificare una sola linea.

È bene tenere presente che le macros non sono variabili, per cui istruzioni del tipo `++SIZE` o `SIZE = 25` non hanno senso. Infatti la prima viene vista dal compilatore, dopo che il file è stato processato dal preprocessore, come `++10` e la seconda come `10 = 25`, che ovviamente non hanno nessun senso. Per limitare la possibilità di errori di questo tipo, sebbene non sia necessario, è pratica comune quella di utilizzare solo lettere maiuscole per i nomi delle macros in modo da poterle identificare facilmente.

Il corpo della macro può essere praticamente qualsiasi cosa perchè il preprocessore non interpreta i comandi ne controlla la sintassi del linguaggio C ma si limita a sostituire il nome della macro con la `string-of-text` che definisce il suo corpo. È quindi possibile ad esempio sostituire parole o caratteri riservati del linguaggio:

```
#define BEGIN {
#define END   }
int main(void)
BEGIN
  int i;

  if (i > 0)
  BEGIN
    printf("Ciao!\n");
  END

  return (0);
END
```

oppure sostituire parti di istruzioni:

```
#define FOR_ALL_I for (i = 0; i < SIZE; ++i)

...
  FOR_ALL_I {
    a[i] = 0;
  }
```

Sebbene ciò sia possibile questi sono chiaramente esempi di programmazione piuttosto questionabili perchè rendono il programma difficilmente comprensibile.

Le linee contenenti le direttive del preprocessore sono riconosciute dal preprocessore prima di effettuare le espansioni delle macros, di conseguenza se l'espansione di una macro produce qualche cosa che assomigli ad una direttiva questa verrà ignorata. Ad esempio

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

non ha come conseguenza quello di includere il file di header `stdio.h` nel programma ma quello di passare la `string-of-text`

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

al compilatore, che ovviamente non sarà affatto contento.

Siccome il preprocessore non controlla che l'espansione delle macros produca espressioni sintatticamente corrette il suo uso può a volte causare errori inaspettati e soprattutto difficili da identificare. Ad esempio il seguente programma

```
1  #define SIZE 2 ** 4
2
3  int main(void)
4  {
5    int      indice;
6    int array[SIZE];
7    for (indice = 0; indice < SIZE; indice++)
8      {
9        array[indice] = 0;
10     }
11
12   return 0;
13 }
```

produce un errore in compilazione alla riga 6. Tuttavia l'errore è nella riga 1 poichè il preprocessore espande la macro `SIZE` in `2 ** 4`, per cui il compilatore C si trova l'istruzione

```
int array[2 ** 4];
```

che contiene l'operatore illegale "`**`" e quindi produce un messaggio di errore.

Per vedere l'effetto del preprocessore si può utilizzare la *flag* "`-E`" del compilatore:

```
$ cc -E program_file.c
```

in modo che il file processato non venga inviato al compilatore ma al terminale. Nel caso dell'esempio in questione si avrà un risultato simile

```
$ cc -E wrong_define.c
# 1 "wrong_define.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "wrong_define.c"


int main(void)
{
  int indice;
  int array[2 ** 4];
  for (indice = 0; indice < 2 ** 4; indice++)
    {
      array[indice] = 0;
    }

  return 0;
}
```

da cui risulta evidente che l'errore in linea 6 è causato dall'espansione della macro `SIZE`.

Nell'esempio appena discusso l'errore viene segnalato dal compilatore, ma non sempre si è così fortunati. Il seguente programma ad esempio viene compilato senza errori:

```
#include <stdio.h>

#define I_MIN   3
#define I_MAX   8
#define DATA    I_MAX - I_MIN + 1

int main(void)
{
  printf("Numero dati            : %d\n", DATA);
  printf("Numero dati al quadrato: %d\n", DATA * DATA);
  return 0;
}
```

ma quando viene eseguito produce il seguente risultato

```
Numero dati            : 6
Numero dati al quadrato: 11
```

La causa dell'errore è anche in questo caso da ricercarsi nelle definizioni delle macros, ed in particolare nella definizione della macro `DATA` poichè il preprocessore espande `DATA * DATA` come

$$\texttt{DATA} * \texttt{DATA} \Rightarrow 8 - 3 + 1 * 8 - 3 + 1 = 11$$

Questo errore può essere più difficile da risolvere del precedente. In questi casi l'uso del *flag* "`-E`" può semplificare enormemente la ricerca e la correzione dell'errore, infatti:

```
$ cc -E bad_define.c
# 1 "bad_define.c"
# 1 "/usr/include/stdio.h" 1 3
  .....
# 2 "bad_define.c" 2
int main(void)
{
  printf("Numero dati            : %d\n", 8 - 3 + 1);
  printf("Numero dati al quadrato: %d\n", 8 - 3 + 1 * 8 - 3 + 1);
  return 0;
}
```

da cui l'origine dell'errore risulta chiaramente. In questo caso l'errore può essere corretto facilmente con l'uso delle parentesi:

```
#include <stdio.h>

#define I_MIN   3
#define I_MAX   8
#define DATA    (I_MAX - I_MIN + 1)

int main(void)
{
```

```
    printf("Numero dati             : %d\n", DATA);
    printf("Numero dati al quadrato: %d\n", DATA * DATA);
    return 0;
}
```

in modo che `DATA * DATA` venga espanso dal preprocessore come

$$\texttt{DATA} * \texttt{DATA} \Rightarrow (8-3+1)*(8-3+1) = 36$$

ed infatti adesso quando il programma viene eseguito produce il risultato corretto

```
Numero dati             : 6
Numero dati al quadrato: 36
```

Un errore piuttosto comune è quello di inserire un segno di uguaglianza "`=`" nella definizione:

```
#include <stdio.h>
#define MAX = 10

int main(void)
{
  int counter;

  for (counter = MAX; counter > 0; counter--) {
    printf("Ciao!\n");
  }
  return 0;
}
```

In questo caso la macros `MAX` viene sostituita con "`= 10`" generando un errore.

Un altro errore piuttosto comune, sopratutto le prime volte, è quello di aggiungere il "`;`" alla fine della direttiva come se si trattasse di un'istruzione in linguaggio C, ad esempio

```
#define MAX  10;
```

Anche in questo caso è molto probabile che l'espansione della macros generi un errore.

Macros con parametri

È possibile definire macros più complesse specificando una lista di nomi di parametri formali separati dalla virgola e racchiusi dalle parentesi tonde:

```
#define macro_name(name_1, name_2,...) string-of-text  /* comment */
```

Tra il nome della macro `macro_name` e la parentesi non vi deve essere nessuno spazio altrimenti la macro viene interpretata come una macro senza parametri formali il cui corpo inizia con la parentesi. Il corpo della macro può contenere parentesi purché correttamente chiuse ed aperte. I nomi dei parametri formali devono essere tutti diversi.

Il seguente è un esempio di definizione di macro con parametri

```
#define POW2(x)  ((x) * (x))
```

In questa definizione `x` è il nome di un parametro formale che nell'espansione della macro viene sostituito con l'argomento della macro `POW2`. Ad esempio

```
POW2(a + b)
```

viene espanso dal preprocessore in

```
((a + b) * (a + b))
```

Come nel caso delle macros senza argomenti, anche in questo caso la sostituzione viene effettuata senza controllare la correttezza della sintassi.

Osserviamo che se la macro fosse stata definita con uno spazio tra il nome della macro e la parentesi

```
#define POW2 (x)  ((x) * (x))      /* Errato !!! */
```

`x` non sarebbe stato interpretato come nome di un parametro formale ma come parte del corpo della macro, per cui `POW2(a + b)` sarebbe stato espanso in

```
(x)  ((x) * (x))(a + b)
```

che molto probabilmente non è esattamente il risultato voluto.

Nella definizione della macro `POW2` si è fatto un uso notevole di parentesi per evitare che l'espansione della macro produca espressioni valutate in ordine diverso da quello voluto. Infatti se ad esempio `POW2` fosse stata definita senza parentesi

```
#define POW2 (x)  x * x
```

il risultato dell'espansione di `POW2(a + b)` sarebbe stata l'espressione

```
a + b * a + b
```

che viene valutata diversamente da

```
(a + b) * (a + b)
```

La definizione

```
#define POW2 (x)  (x) * (x)
```

risolverebbe questo problema ma non

```
6 / POW2 (3)
```

che viene espanso in

```
6 / (3) * (3)
```

che è diverso dal risultato voluto

```
6 / ((3) * (3))
```

Come norma generale quando si definiscono macros bisogna fare molta attenzione a come queste vengono espanse. Se le macros hanno argomenti questi andrebbero sempre messi tra

parentesi nel corpo della macro. Inoltre se il corpo della macro è sintatticamente un'espressione, come nel caso della nostra macro `POW2`, anche questo va racchiuso tra parentesi.

Le macros con parametri vengono spesso usate per sostituire le chiamate a funzioni con delle istruzioni "in linea". Ad esempio le seguenti macros permettono di determinare il valore minore e il valore massimo tra due

```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
#define MAX(x,y) ((x) > (y) ? (x) : (y))
```

In questo modo espressioni del tipo

```
z_min = MIN(x,y);
z_max = MAX(x,y);
```

vengono espanse dal preprocessore in

```
z_min = ((x) < (y) ? (x) : (y))
z_max = ((x) > (y) ? (x) : (y))
```

Di nuovo le parentesi sono necessarie per non avere sorprese nell'espansione.

Il C99 permette di definire macros con un numero variabile di parametri, in questo caso la lista dei parametri viene sostituita da tre punti "`...`". Non discuteremo questa possibilità e considereremo solo macros con un numero fissato di parametri.

In genere i compilatori C permettono di definire le macros *on-the-fly* direttamente dalla linea di comando di compilazione utilizzando il *flag* "`-D`". Ad esempio il comando

```
$ cc -DSIZE=10 -DDEBUG program.c
```

è equivalente ad aggiungere all'inizio del file sorgente `program.c` le direttive

```
#define SIZE 10
#define DEBUG
```

Anche macros con parametri possono essere definite in questo modo, il comando

```
$ cc -D"POW2(x)=((x)*(x))" program.c
```

ad esempio è equivalente ad aggiungere la direttiva

```
#define POW2(x)   ((x)*(x))
```

L'uso dei doppi apici "`"`" è in questo caso necessario. Chiaramente questo modo di definire la macros non è sempre dei più comodi.

2.18.3. Comando `#undef`

Il comando

```
#undef macro_name
```

elimina la definizione della macro `macro_name` che viene dimenticata da preprocessore e quindi non può essere più utilizzata nel resto del file. Nel caso di macros con parametri questi

ultimi non vanno specificati. Se la macro `name_macro` non era definita non viene prodotto nessun messaggio di errore. Una volta che una macro è stata cancellata può essere definita nuovamente con il comando `#define`.

Lo Standard C permette di definire una stessa macro più volte nello stesso file a patto che la definizione sia la stessa, inclusi gli spazi nelle stesse posizioni. Il carattere utilizzato per lo spazio, *space* o `tab` o il numero di spazi può essere differente. Ad esempio le due definizioni

```
#define MACRO 1
#define MACRO /* commento == spazio bianco */ 1
#define MACRO     1
```

sono equivalenti mentre le seguenti

```
#define POW2(x) ((x) * (x))
#define POW2(x) ((x)*(x))
#define POW2(y) ((y) * (y))
```

non lo sono poiché nella prima mancano gli spazi mentre nella seconda cambia il nome del parametro. Alcuni preprocessori non sono così rigidi ed accettano queste definizioni come ridefinizioni della macro. Tuttavia questo comportamento non è standard per cui se si vuole ridefinire una macro e si vogliono evitare problemi di compatibilità conviene sempre utilizzare il comando `#undef` per cancellare la definizione la vecchia definizione prima di ridefinire una macro. Così ad esempio le seguenti ridefinizioni

```
#define POW2(x) ((x) * (x))
#undef  POW2
#define POW2(x) ((x)*(x))
#undef  POW2
#define POW2(y) ((y) * (y))
```

non creano nessun problema. Ricordiamo che con il comando `#undef` gli eventuali parametri della macro non vanno specificati.

Analogamente alla definizione in genere è possibile *un*definire una macro *on-the-fly* direttamente dalla linea di comando di compilazione utilizzando il *flag* "`-U`" del compilatore. Ad esempio il comando

```
$ cc -UDEBUG program.c
```

è equivalente ad aggiungere all'inizio del file sorgente `program.c` la direttiva

```
#undef DEBUG
```

Questa possibilità risulta particolarmente utile nella compilazione condizionata, come vedremo tra poco.

2.18.4. Comando `#include`

Abbiamo già incontrato più volte questo comando del preprocessore. Il comando `#include` permette di inserire il contenuto di un altro file nel file sorgente. Il file viene inserito al posto dalla linea del comando `#include` e letto come parte integrante del file sorgente. Le due forme principali del comando `#include` sono

```
#include <filename>
#include "filename"
```

La differenza nelle due forme è nella ricerca del file da inserire. Nella prima forma con `<...>` il file `filename` viene cercato nelle directories standard dove usualmente risiedono files di sistema. In generale queste dipendono dal sistema, sui sistemi UNIX generalmente i files di sistema sono nella directory `/usr/include/`.

Anche la forma con `"..."` cerca nelle directories standard, ma solo dopo aver cercato in altre directories "locali". La lista delle directories locali, come quella delle directories standard, dipende dal sistema ma in ogni caso include la directory locale in cui si sta compilando il programma. Di conseguenza l'uso principale della forma con `"..."` è per inserire files scritti dal programmatore, mentre quella con `<...>` per inserire files di sistema. Ad esempio

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

include il file di header di sistema `stdio.h` che si deve trovare nelle directories standard, mentre

```
#include "pezzo.c"
```

include il file `pezzo.c` che generalmente si deve trovare nella directory in cui si sta compilando il programma. Il nome del file da includere può contenere informazioni sul percorso (*path*), sia in forma relativa che assoluta, per raggiungere la directory con il file da includere. In generale la sintassi per specificare le directories dipende dal sistema operativo, ad esempio su sistemi UNIX

```
#include "../pezzo.c"
```

indica che il file si trova nella directory subito "sopra" (path relativo) mentre

```
#include "/usr/include/stdio.h"
```

indica che il file si trova nella directory `/usr/include` (path assoluto).

Un file incluso con il comando `#include` può a sua volta includere altri files che a loro volta possono includerne altri e così via. Il C89 richiede che il numero massimo di `#include` successivi sia almeno otto, mentre il C99 porta questo limite a quindici. In generale il numero di `#include` successivi permessi dipende dal sistema. Sebbene non vi siano restrizioni sul contenuto dei files inclusi usualmente questi contengono solo definizioni e/o dichiarazioni ma non istruzioni.

Nota. La sintassi dei nomi dei files può dipendere dal sistema. Lo Standard C richiede che in ogni caso possano essere inclusi con il comando `#include` files il cui nome è formato da lettere e digits decimali, con il primo carattere una lettera, seguiti da un punto "`.`" e da una lettera. Il C89 richiede che vi siano fino a cinque caratteri prima del punto, mentre il C99 ne richiede otto. In realtà tutti i sistemi sono molto più flessibili.

2.18.5. Comandi `#if`, `#ifdef`, `#ifndef` e `#endif`

Le istruzioni `#if`, `#ifdef` ed `#ifndef` permettono di includere od escludere parti del programma a seconda del valore di alcune condizioni rendendo quindi possibile una compilazione condizionata. La forma di una direttiva condizionale del preprocessore è

```
#if      constant-expression
#ifdef   macro_name
#ifndef  macro_name
```

dove `constant-expression` è un'espressione il cui valore deve essere intero. Se il valore è 0 la condizione è considerata *falsa* mentre per ogni valore non nullo è considerata *vera*. Se nell'espressione figurano macros il preprocessore le espande prima di valutare il valore dell'espressione. Nel caso di `#ifdef` la condizione è *vera* se la macro `macro_name` è stata definita con il comando `#define`, mentre con il comando `#ifndef` la condizione è *vera* se la macro `macro_name` non è definita sia perchè è stata cancellata con il comando `#undef` sia perchè non è mai stata definita.

Nel caso che la condizione sia valutata *vera* le linee di programma che seguono la direttiva condizionale fino alla linea che contiene la direttiva

```
#endif
```

sono incluse nel file sorgente preprocessato generato dal preprocessore, e quindi nella compilazione del programma.

La possibilità di poter effettuare una compilazione condizionata risulta utile in molti casi. Supponiamo ad esempio di voler escludere alcune linee del programma senza cancellarle perchè dovranno poi essere reinserite successivamente. Un modo di procedere è quello di commentare tutte le linee non desiderate con i delimitatori "`/*`" e "`*/`", il che però può risultare piuttosto scomodo se le linee sono molte. Alternativamente, e più semplicemente, si può utilizzare il comando `#if` del preprocessore:

```
#if 0
linee da eliminare dalla compilazione
#endif
```

In questo modo le linee del programma tra le direttive `#if` ed `#endif` vengono escluse dalla compilazione. Quando poi sarà necessario reinserirle basterà cambiare "`#if 0`" con "`#if 1`", o con "`#if` seguito da un qualsisi valore non nullo, ed il gioco è fatto.

Chiaramente il procedimento può essere invertito ed usato per includere durante la fase di sviluppo del programma linee di controllo, linee che poi devono essere eliminate nella versione finale. In questo caso, sopratutto se le linee vanno inserite in più parti del file sorgente, è più comodo utilizzare il comando `#ifdef` ed una macro, come mostrato qui di seguito. Supponiamo che in fase di sviluppo del programma ci interessi conoscere il valore che la variabile `i` assume in determinato punto del programma. Possiamo allora inserire nel punto desiderato le linee

```
#ifdef DEBUG
    printf(" Il valore della variabile i e': %d\n", i);
#endif /* DEBUG */
```

Non è necessario mettere il commento "`/* DEBUG */`" dopo `#endif`, però può essere utile per sapere a quale direttiva condizionale fa riferimento. Se all'inizio del programma viene messa la direttiva

```
#define DEBUG
```

la macro `DEBUG` è definita e quindi la condizione `#ifdef DEBUG` risulta *vera* per cui l'istruzione di output viene passata dal preprocessore C al compilatore C. Non è necessario che la macro `DEBUG` abbia un corpo affinchè la condizione sia *vera* è sufficiente che sia stata definita.

Se invece all'inizio del programma vi è la direttiva

```
#undef DEBUG
```

l'istruzione di output non viene passata al compilatore C e quindi il valore della variabile `i` non viene stampato.

Più linee di controllo possono essere messe in diverse parti del file utilizzando questa procedura, queste saranno poi attivate o disattivate a seconda della definizione della macro `DEBUG`.

La direttiva `#undef DEBUG` non è in realtà necessaria poichè in mancanza di una definizione esplicita con il comando `#define` la macro `DEBUG` risulta non definita. Tuttavia è consigliato metterla perchè rende il programma di più facile lettura, ad esenpio in mancanza di un comando `#define` o `#undef` non è immediato risalire se la macro non è stata definita volontariamente o se è una dimenticanza. Inoltre permette di conoscere più facilmente quali macros sono utilizzate nel file.

Se si desidera è possibile attivare o disattivare il debugging direttamente dal comando di compilazione utilizzando le *flags* "`-D`" e "`-U`" del compilatore senza quindi dover ogni volta cambiare il file sorgente.

Il comando `#ifndef` viene utilizzato spesso per fornire una definizione di `default` di una macro. Nell'esempio dell'array la macro `SIZE` fornisce la dimensione dell'array. Se si è scelto di definirla utilizzando il *flag* "`-D`" del compilatore può capitare di dimenticarsene ed in questo caso è molto probabile che la compilazione fallisca. Questo è un caso fortunato, ma può anche capitare che la mancata definizione di una macro non produca effetti così evidenti. Per mettersi al riparo da queste situazioni è bene fornire una definizione di default da utilizzarsi nel caso venga a mancare la definizione *on-the-fly*. Questo può essere ottenuto facilmente con il comando "`#ifndef`, ad esempio le direttive

```
#ifndef SIZE
#define SIZE 10   /* valore di default */
#endif
```

definiscono la macro `SIZE` nel caso non ne venga data precedentemente una definizione.

Un altro uso frequente di questa istruzione è per evitare di avere definizioni multiple di una stessa macro o inclusioni multiple dello stesso file. Supponiamo ad esempio che il file sorgente contenga le direttive

```
#include "header_1.h"
#include "header_2.h"
```

e che entrambi i files di header includano il file `definition.h`. Chiaramente questo file viene incluso due volte, una volta da `header_1.h` ed una volta da `header_2.h`. Per evitare la doppia inclusione si può definire una macro che controlli se il file è già stato incluso introducendo nel file `definition.h` le direttive

```
#ifndef DEF_INCLUDED
#define DEF_INCLUDED
#endif
```

che definiscono la macro `DEF_INCLUDED` la prima volta che `definition.h` viene incluso. Per evitare inclusioni multiple il file `definition.h` deve essere incluso nei files `header_1.h` e `header_2.h`. con le direttive

```
#ifndef DEF_INCLUDED
#include "definition.h"
#endif
```

In questo modo la prima inclusione del file `definition.h` definisce la macro `DEF_INCLUDED` e tutte le eventuali inclusioni successive vengono saltate.

2.18.6. Comandi `#else` e `#elif`

I comandi `#else` e `#elif` introducono alternative nella compilazione condizionata. Ad esempio per sapere se il debuggung è attivo o no nel nostro esempio precedente potremmo inserire nel file sorgente le direttive

```
#ifdef DEBUG
    printf(" Debugging abilitato\n");
#else           /* DEBUG */
    printf(" Debugging non abilitato\n");
#endif         /* DEBUG */
```

in modo che sia inviato sullo `stdout` un messaggio o l'alttro a seconda della definizione della macro `DEBUG`.

Il comando `#elif` è equivalente ad una costruzione `else-if` e permette di avere più alternative. Questo comando è usato nella forma

```
#if constant-expression_1
  gruppo di linee 1
#elif constant-expression_2
  gruppo di linee 2
...
#elif constant-expression_n
  gruppo di linee n
#else
  ultimo gruppo di linee
#endif
```

Se `constant-expression_1` è *vera* le linee del gruppo 1 vengono incluse per la compilazione, altrimenti il preprocessore controlla `constant-expression_2` e se è *vera* sono le linee del gruppo 2 ad essere incluse. Se invece anche `constant-expression_2` è *falsa* si passa a

`constant-expression_3` è così via. Nel caso in cui tutte le espressioni siano `false` viene incluso l'ultimo gruppo di linee che seguono la direttiva `#else`. Se invece la direttiva `#else` non viene fornita e tutte le espressioni sono false nessuna linea viene inclusa.

2.18.7. Operatore `defined`

L'operatore del preprocessore `defined` può essere usato solo con i comandi `#if` e `#elif`. L'operatore può prendere una delle due forme

```
defined macro_name
defined(macro_name)
```

ed il suo valore è `1` se la macro `macro_name` è definita o `0` se non lo è. Questo operatore è simile al comando `#ifdef`, infatti

```
#if defined(DEBUG)
```

è perfettamente equivalente a

```
#ifdef DEBUG
```

tuttavia l'operatore `defined` permette di avere costruzioni condizionali più complesse. Ad esempio

```
#if defined(MACRO_A)
  gruppo di linee A
#elif defined(MACRO_B)
  gruppo di linee B
#elif defined(MACRO_C)
  gruppo di linee C
#endif
```

oppure

```
#if defined(MACRO_A) && defined(MACRO_B) && !defined(MACRO_C)
  gruppo di linee
#endif
```

dove "`&&`" e "`!`" sono usuali gli operatori logici `AND` e `NOT`. Per cui l'espressione precedente è *vera* solo se sono definite le macros `MACRO_A` e `MACRO_B` ma non la macro `MACRO_C`.

2.19. Esempio: Istogramma di frequenza di un set di dati (Rev. 2.1.1)

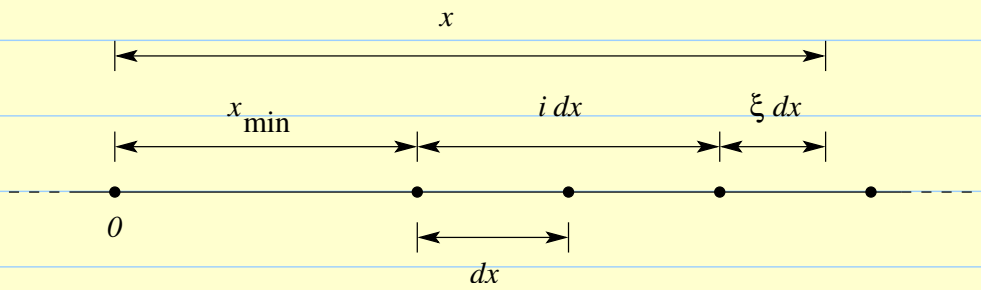
Per illustrare l'uso degli array consideriamo un programma che calcoli l'istogramma in frequenza di un set di dati $[x_1, x_2, x_3, \ldots]$ letti da un file. Vi sono molti modi di scrivere un programma che calcola un istogramma di frequenza a seconda dalle informazioni che si hanno o dal tipo di output che si vuole. Tuttavia l'algoritmo che calcola effettivamente l'istogramma è sostanzialmente sempre lo stesso, per cui per rendere le cose più semplici possibili assumeremo di conoscere solo il nome del file, con i dati scritti uno per riga, ed il nome del file su

cui scrivere l'istogramma. Per rendere il programma un più flessibile il nome dei files viene fornito da tastiera e inoltre si assume che il numero dei dati nel file non sia noto.

L'algoritmo per calcolare un istogramma di frequenza è molto semplice: si divide l'intervallo di variabilità dei dati $[x_{\min}, x_{\max}]$ in N intervalli di ampiezza dx, chiamati *bin* dell'istogramma, e per ogni intervallo si determina quanti dati vi cadono. L'istogramma in frequenza si ottiene a questo punto semplicemente dividendo il numero di dati caduti in ciascun intervallo per il numero totale dei dati utilizzati. La parte più "difficile" dell'algoritmo è quella di determinare per ogni valore di x quale è il bin dell'istogramma corrispondente. Il realtà la cosa non è poi così difficile. Infatti se associamo a ciascuno degli N intervalli in cui è stato diviso l'intervallo $[x_{\min}, x_{\max}]$ un indice i che varia da 0 per il primo intervallo a $N-1$ per l'ultimo è facile convincersi che ogni valore di x nell'intervallo $[x_{\min}, x_{\max})$ si può scrivere come

$$x = x_{\min} + i\,dx + \xi\,dx, \qquad 0 \le \xi < 1$$

dove $dx = (x_{\max} - x_{\min})/N$ e $i = 0, \ldots, N-1$, come mostra la figura seguente



Questa rappresentazione associa al valore $x_{\max}$ l'indice $i = N$ per cui tutti i valori letti per cui vale l'uguaglianza $x = x_{\max}$ sono esclusi dall'istogramma. Vi sono vari modi di ovviare a questo inconveniente, ad esempio basta scegliere un valore di $x_{\max}$ tale che per ogni valore di x si abbia $x < x_{\max}$. Questo è equivalente ad "allargare" in pochino l'ampiezza dx del bin.

Invertendo la rappresentazione precedente, e sfruttando il fatto che l'indice i è un intero, si ottiene la seguente semplice relazione tra il valore di x e l'indice dell'intervallo corrispondente:

$$i = \left[\frac{x - x_{\min}}{dx} \right],$$

dove $[\cdot]$ indica la parte intera del numero, ad esempio $[1.1] = 1$ e $[0.1] = 0$. È facile verificare che $i = 0, \ldots, N-1$ per tutti i valori di x tali che $x_{\min} \le x < x_{\max}$.

Il programma `isto1.c` proposto qui di seguito utilizza questo semplice algoritmo. Prima di passare al programma vero e proprio vi sono però alcuni dettagli aggiuntivi da considerare.

Per prima cosa se i valori di $x_{\max}$ e $x_{\min}$ non sono noti vanno determinati prima di procedere alla costruzione dell'istogramma. La strategia utilizzata dal programma è quella di leggere una prima volta tutti i dati dal file per determinare i valori di $x_{\max}$, $x_{\min}$ e del numero totale di dati, "riavvolgerlo" e rileggere tutti i dati per costruire l'istogramma. Questa strategia può non essere la più veloce perché la lettura da files non è velocissima, però ha il vantaggio che non è necessario memorizzare il valore di tutti i dati nel programma.

L'istogramma può essere facilmente costruito utilizzando un array di tipo `int` i cui elementi contengano il numero di dati che sono caduti in ciascun bin. Se il numero di bin dell'istogramma non è noto, ma ad esempio si fa leggere da tastiera, l'array va dichiarata di dimensione fissata sufficiente grande. Questo scelta non è troppo limitativa perché lascia la possibilità di costruire istogrammi con un numero di bin n_{bin} inferiore alla dimensione dell'array. In altre parole si usa una strategia simile a quella usata per la rappresentazione delle stringhe mediante arrays di tipo `char`. Di conseguenza ciascun bin dell'istogramma è univocamente individuato dall'indice dell'elemento dell'array corrispondente che varierà da 0 per il primo bin a $n_{\text{bin}} - 1$ per l'ultimo bin utilizzato.

Osserviamo che dal momento che l'array occupa memoria anche se non viene completamemte utilizzata la sua dimensione non deve essere troppo grande. Se dovesse risultare insufficiente è possibile aumentarla in seguito, sempre nei limiti del sistema.

Programma: `isto1.c`

```
/*
 * Descrizione : calcola l'istogramma in frequenza di un set di
 *               dati (x_i) letti da in file.
 *
 * Input       : nome del file con di dati
 *               numero di bin dell'istogramma
 *
 * Output      : istogramma su file
 *               informazioni sul terminale
 *
 * Parametri   : MAX_BIN  numero massimo di bin dell'istogramma
 *
 * $yaC-Primer: isto1.c v 1.3 03.02.05 AC $
 */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/* Macros */
#define MAX_BIN 1000                    /* numero max bin */

int main(void)
{
  int            n_data;
  int         ib, n_bin;
  int       yb[MAX_BIN];          /* bin istogramma      */
  double   x_min, x_max;          /* min max dati        */
  double x_mean, x_sigm;          /* media e std. dev.   */
  double            dx;           /* ampiezza bin        */
  double          x_tmp;
  char    file_name[41];          /* nome file           */
  char         line[10];          /* buffer              */
  FILE        *in_file;           /* Input file          */
```

```
  FILE        *out_file;          /* Output file         */

  /* File di input */
  printf("File con i dati      : ");
  fgets(file_name, sizeof(file_name), stdin);
  file_name[strlen(file_name) - 1] = '\0';

  if ( (in_file = fopen(file_name,"r")) == NULL) {
    fprintf(stderr, "\n%s : %s\n\n", file_name, strerror(errno));
    exit(errno);
  }

  /* Apertura file di output */
  printf("File con l'istogramma: ");
  fgets(file_name, sizeof(file_name), stdin);
  file_name[strlen(file_name) - 1] = '\0';

  if ( (out_file = fopen(file_name,"w")) == NULL) {
    fprintf(stderr, "\n%s : %s\n\n", file_name, strerror(errno));
    exit(errno);
  }

  /* bin istogramma */
  while (1) {
    printf("Quanti bin (<= %d): ", MAX_BIN);
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &n_bin);
    if (n_bin > 0 && n_bin <= MAX_BIN) break;
  }

  /* Valore massimo, minimo e numero dei dati */
  if ( fscanf(in_file, "%lf", &x_min) != EOF ) {
    n_data = 1;
    x_max  = x_min;
    x_mean = x_min;
    x_sigm = x_min * x_min;
  }

  while ( fscanf(in_file, "%lf", &x_tmp) != EOF ) {
    ++n_data;

    if ( x_tmp > x_max )      x_max = x_tmp;
    else if ( x_tmp < x_min ) x_min = x_tmp;

    x_mean += (x_tmp         - x_mean) / (double) n_data;
    x_sigm += (x_tmp * x_tmp - x_sigm) / (double) n_data;
  }

  /* Varianza dadi */
  x_sigm -= x_mean * x_mean;

  /* Informazini sui dati */
```

```
  printf("\nLetti %d dati\n", n_data);
  printf("x_min: %.4f \t x_max: %.4f \t dx: %.4f\n",
          x_min, x_max, dx);
  printf("media: %.4f  \t    deviazione std.: %.4f\n\n",
         x_mean, sqrt(x_sigm));

  /* Inizializzazione istogramma */
  for (ib = 0; ib < n_bin; ++ib) yb[ib] = 0;

  /* Largezza bin  */
  dx = 1.01*(x_max - x_min) / (double) n_bin;

  /* file di input all'inizio */
  rewind(in_file);

  /* Inserimento dati nei bin */
  while (fscanf(in_file, "%lf", &x_tmp) != EOF ) {
    ib      = (int) ( (x_tmp - x_min) / dx);
    ++yb[ib];
  }
  fclose(in_file);

  /* Scrittura dell'istogramma */
  for (ib = 0; ib < n_bin; ++ib) {
    fprintf(out_file, "%f \t %f \t %f\n",
            x_min + dx * (0.5 + (double) ib ),
            (double) yb[ib] / (double) n_data,
            (double) yb[ib] / (n_data * dx)
            );
  }
  fclose (out_file);

  return 0;
}
```

Note sul programma: `isto1.c`

- `#define` MAX_BIN 1000

Questa è un'istruzione per il preprocessore che definisce la *macro* `MAX_BIN`. Quando il programma viene compilato il preprocessore sostituisce ogni occorrenza di `MAX_BIN` con `1000` prima di passare il codice al compilatore. Così ad esempio l'istruzione

```
int yb[MAX_BIN];
```

viene vista dal compilatore vero e proprio come

```
int yb[1000];
```

L'uso di una *macro* permette da un lato di poter cambiare facilmente il numero massimo di bins, basta cambiare in una linea e non ovunque serva, e dall'altro di soddisfare la

richiesta del compilatore C di conoscere l'esatta dimensione degli arrays. Ricordiamo che nel C89 non sono permessi arrays di dimensione variabile.

```
FILE         *in_file;          /* Input file          */
FILE        *out_file;          /* Output file         */
```

Si usano due streams: `in_file` per il file di input con i dati e `out_file` per il file di output con l'istogramma. I nomi di entrambi i files sono letti dallo stream standard di input `stdin`. In realtà dal momento che non si usano i due files contemporaneamente si sarebbe potuto utilizzare anche un solo stream. L'uso di due streams rende però il programma più chiaro.

Il nome dei files viene letto utilizzando la funzione `fgets()` come discusso in dettaglio nel programma per l'interpolazione lineare.

```
while (1) {
  printf("Quanti bin (<= %d): ", MAX_BIN);
  fgets(line, sizeof(line), stdin);
  sscanf(line, "%d", &n_bin);
  if (n_bin > 0 && n_bin <= MAX_BIN) break;
}
```

Si chiede il numero di bins dell'istogramma desiderato informando quale è il numero massimo ammesso. Il ciclo `while` assicura che il numero letto sia un numero valido, ossia positivo e non più grande della dimensione `MAX_BIN` dell'array. Nel ciclo `while` la condizione logica è sempre *vera* per cui il ciclo viene interrotto con l'istruzione `break` se il valore fornito per `n_bin` soddisfa le richieste. In caso contrario il ciclo ricomincia dall'inizio richiedendo un nuovo valore per `n_bin`. Osserviamo che l'ordine nella condizione logica è rilevante, infatti siccome in un `AND` logico l'operando a destra non viene valutato se l'operando a sinistra è *falso* l'espressione logica

```
n_bin <= MAX_BIN && n_bin > 0
```

è sempre *vera* se `n_bin` è negativo!

```
if ( fscanf(in_file, "%lf", &x_min) != EOF ) {
  n_data = 1;
  x_max  = x_min;
  x_mean = x_min;
  x_sigm = x_min * x_min;
}

while ( fscanf(in_file, "%lf", &x_tmp) != EOF ) {
  ++n_data;

  if ( x_tmp > x_max )      x_max = x_tmp;
  else if ( x_tmp < x_min ) x_min = x_tmp;

  ...
}
```

Per calcolare il valore minimo e massimo dei dati senza conoscerne a priori l'intervallo di variabilità si legge il primo dato dal file e si assegna il suo valore sia a `x_min` che a `x_max`. I dati successivi vengono letti una alla volta ed il valore confrontato con `x_max`. Se il valore è maggiore di `x_max` questo diventa il nuovo `x_max`, altrimenti il valore viene confrontato con `x_min` e se è minore di `x_min` questo diventa il nuovo `x_min`. Se il valore cade tra `x_min` e `x_max` i valori degli estremi non vengono modificati.

- ```
  x_mean += (x_tmp          - x_mean) / (double) n_data;
  x_sigm += (x_tmp * x_tmp - x_sigm) / (double) n_data;
  ```

Contestualmente alla ricerca del valore minimo e massimo dei dati il programma calcola anche il valore medio e la varianza dei dati. Questo è fatto utilizzando la formula *run-time*

$$\langle x \rangle_n = \langle x \rangle_{n-1} + (x - \langle x \rangle_{n-1})$$

dove $\langle x \rangle_n = (1/n) \sum_{i=1}^{n} x_i$. Questa relazione si verifica facilmente sostituendo le espressioni esplicite di $\langle x \rangle_n$ e $\langle x \rangle_{n-1}$.

- ```
  for (ib = 0; ib < n_bin; ++ib) yb[ib] = 0;
  ```

I valori dei bin vengono inizializzati a zero perché all'inizio non vi sono ancora stati inseriti dati.

- ```
  dx = 1.01*(x_max - x_min) / (double) n_bin;
  ```

L'intervallo viene "ingrandito" dell' 1% in modo che l'indice del bin `ib` valga `0` per $x =$ `x_min` e `n_bin-1` per $x =$ `x_max`. Se non si usa questo espediente, od uno simile, il valore $x =$ `x_max` cadrebbe nel bin di indice `n_bin`. Questo non è un problema se `n_bin` è minore di `MAX_BIN` ma potrebbe diventarlo nel caso siano uguali perché si uscirebbe dallo spazio di memoria riservato all'array.

- ```
  rewind(in_file);
  ```

Il file di input viene "riavvolto" per poter rileggere i dati dall'inizio.

- ```
  while (fscanf(in_file, "%lf", &x_tmp) != EOF ) {
    ib      = (int) ( (x_tmp - x_min) / dx);
    ++yb[ib];
  }
  ```

I dati vengono letti dal file uno alla volta e per ciascuno si determina l'indice `ib` del bin associato all'intervallo in cui cade il suo valore. Una volta determinato il bin si aumenta di `1` il contatore `yb[ib]` del numero di dati che caduti nel bin, ossia l'elemento di indice `ib` dell'array `yb` che contiene l'istogramma.

- ```
  for (ib = 0; ib < n_bin; ++ib) {
    fprintf(out_file, "%f \t %f \t %f\n",
            x_min + dx * (0.5 + (double) ib ),
            (double) yb[ib] / (double) n_data,
            (double) yb[ib] / (n_data * dx)
           );
  }
  ```

L'istogramma viene scritto sul file identificato da `out_file` utilizzando come valore x del bin il valore centrale, $\xi = 0.5$ nella rappresentazione precedente. Altre scelte sono ovviamente possibili. I valori dell'istogramma sono scritti con due "normalizzazioni" differenti, la prima assicura la somma dei valori dell'istogramma sia uguale 1 la seconda che l'integrale sia uguale a 1. In altre parole la prima è la frequenza la seconda la densità di probabilità.

Quando il programma viene eseguito si ha

```
File con i dati      : isto.dat
File con l'istogramma: isto.his
Quanti bin (<= 1000): 20

Letti 1000 dati
x_min: 35.8570   x_max: 66.2980          dx: 1.5373
media: 50.2143              deviazione std.: 5.0043
```

In questo esempio il file `isto.dat` contiene i dati, mentre l'istogramma su 20 bins viene scritto sul file `isto.his`.

Test del programma: Per controllare i risultati del programma si utilizza un file con dei dati di cui si conosce il risultato, ad esempio con una distribuzione uniforme o gaussiana.

2.20. Funzioni (Rev. 2.1.2)

Spesso nella scrittura di un programma accade che un gruppo di istruzioni debba essere ripetuto più volte in parti differenti del programma. Altre volte invece può risultare comodo poter separare uno o più gruppi di istruzioni dalla parte principale del programma, ad esempio per migliorarne la lettura o semplificarne lo sviluppo o migliorarne la portabilità. Tutte queste operazioni sono notevolmente semplificate dall'uso delle funzioni che permettono di raggruppare insieme un gruppo di istruzioni e di poterle richiamare in un modo molto semplice e compatto. Le funzioni giocano un ruolo fondamentale nella programmazione in linguaggio C, più che in altri linguaggi. Abbiamo già incontrato diverse funzioni, prima fra tutte la funzione `main()`. Questa funzione ha un ruolo particolare e deve esistere in ogni programma in linguaggio C perché è da questa funzione che inizia l'esecuzione del programma. Tutte le altre funzioni sono chiamate direttamente od indirettamente a partire dalla funzione `main()`. Il massiccio uso delle funzioni rende i programmi scritti in linguaggio C molto flessibili e facilmente adattabili a sistemi ed esigenze diverse, ed è per questi motivi che il linguaggio C utilizza funzioni anche per effettuare operazioni di base, come ad esempio le operazioni di Input/Output, piuttosto che istruzioni primarie del linguaggio.

Da un punto di vista concettuale una funzione è un oggetto più complesso di quelli incontrati fino ad ora e può essere pensato come un oggetto che prende dei valori attraverso dei parametri formali, li elabora ed eventualmente restituisce un risultato. In un certo senso una funzione è come un programma, non a caso tutti i programmi in C sono una funzione: la funzione `main()`.

Le funzioni possono essere introdotte in due modi: il primo è la definizione di funzione che crea il tipo funzione definendone i parametri, il valore restituito, ed il corpo, ossia l'insieme di istruzioni che formano la funzione, ed infine gli associa un identificatore o nome. Il secondo è una dichiarazione di funzione che si riferisce ad un tipo funzione definito da qualche altra parte nello stesso file sorgente od in un file sorgente differente.

2.20.1. Definizione del tipo funzione

Nello Standard C la forma della definizione del tipo funzione di tipo `T` con `n` parametri di tipo `T_1`, ..., `T_n` è

```
type (function_name)(type_1 parameter_1, ... , type_n parameter_n)
{
  statement_1;
  statement_2;
  ....
  statement_m;
  return expression;
}
```

dove

`type`	⇒	tipo `T` del valore restituito dalla funzione
`function_name`	⇒	identificatore o nome della funzione
`type_i`	⇒	tipo `T_i` dei parametri della funzione
`parameter_i`	⇒	parametri formali della funzione
`{...}`	⇒	corpo della funzione
`statement_i`	⇒	istruzioni che formano il corpo della funzione
`return`	⇒	termina l'esecuzione della funzione. Può comparire più volte nel corpo della funzione.
`expression`	⇒	espressione il cui valore viene restituito dalla funzione, omessa nel caso la funzione non restituisca nessun valore, funzione di tipo `void`.

Le parentesi "`(`", "`)`" intorno all'identificatore non sono obbligatorie per cui in genere, a meno che non siano necessarie per la corretta interpretazione della definizione, vengono omesse.

Analizzando la dichiarazione di funzione si nota che questa è composta da due parti: la prima è la dichiarazione di funzione

```
type (function_name)(type_1 parameter_1, ... , type_n parameter_n)
```

che specifica il nome, il tipo del valore restituito, ed il numero ed il tipo dei parametri della funzione. La seconda parte

```
{
  statement_1;
  statement_2;
  ....
  statement_m;
```

```
  return expression;
}
```

è il corpo della funzione formato dall'insieme delle istruzioni che vengono eseguite ogni volta che la funzione viene chiamata.

Questa forma di definizione di funzione introdotta dallo Standard C viene chiamata anche definizione in forma di *prototipo* perché nella dichiarazione oltre al nome ed al tipo della funzione viene data la lista dei suoi parametri con il rispettivo tipo. La dichiarazione fornisce quindi il prototipo della funzione.

Lo Standard C accetta anche la definizione di funzione nella forma del Traditional C in cui il tipo dei parametri viene specificato dopo la dichiarazione, ma prima del corpo, in un ordine qualsiasi.

Il seguente esempio mette a confronto le due forme di definizione

```
int f(double x, double y)          /* forma prototipo     */
{
  ...
}

int f(x, y)                        /* forma tradizionale */
  double x;
  double y;
{
  ...
}
```

Sebbene sia possibile utilizzare entrambe le forme, e sotto certe condizioni mischiare le due, l'uso della forma tradizionale è fortemente scoraggiata. I motivi non sono solo di stile di programmazione ma anche di controllo ed efficienza. Infatti l'uso della definizione in forma di prototipo permette al compilatore di effettuare controlli sui parametri per generare chiamate di funzioni più efficienti, riducendo al contempo la probabilità di errori di programmazione. In questo Primer non faremo uso delle definizioni di funzione in forma tradizionale.

Il tipo del valore restituito dalla funzione può essere un tipo `T` qualsiasi con l'esclusione del tipo array e del tipo funzione. Detto in altre parole una funzione non può restituire un array o una funzione, anche se può restituire un puntatore ad array o funzione. Avremo modo di riconsiderare questo punto più avanti. Se la funzione non restituisce nessun valore il tipo `T` è `void`. È un errore utilizzare una funzione di tipo `void` in un contesto che richiede un valore ed in questi casi il risultato è in genere indefinito. Nello Standard C89 se il tipo di una funzione non viene specificato si assume che la funzione sia di tipo `int` e quindi, se questo è il caso, nella definizione l'indicazione esplicita del tipo può essere omessa. La stessa regola si applica anche ai parametri della funzione, ossia se non viene specificato il tipo di un parametro questo viene considerato per *default* di tipo `int`. Tuttavia se nessun tipo è specificato non è immediato sapere se la funzione e/o il parametro sia realmente di tipo `int` oppure vi è stata una dimenticanza per cui si consiglia di dichiarare sempre esplicitamente il tipo delle funzioni e/o dei parametri anche nel caso in cui questi siano realmente di tipo `int`. Lo Standard C99 richiede che il tipo della funzione e dei parametri sia sempre specificato.

Infine sebbene non sia obbligatorio può essere utile inserire un commento prima della definizione di funzione con informazioni sulla funzione. Questo piccolo accorgimento spesso rende un programma molto più facilmente leggibile.

Istruzione `return`

L'istruzione `return` viene usata per terminare l'esecuzione della funzione. L'istruzione `return` prende la forma

```
return expression_opt;
```

dove `expression_opt` è un'espressione opzionale. Quando viene incontrata un'istruzione `return` l'esecuzione della funzione viene interrotta e l'esecuzione del programma continua con l'istruzione che ha chiamato la funzione. Se dopo l'istruzione `return` vi è un'espressione questa viene valutata ed il suo valore restituito come valore dalla funzione. Ad esempio se la funzione viene terminata con l'istruzione

```
return 3 * i;
```

l'espressione "`3 * i`" viene valutata utilizzando il valore della variabile `i` ed il valore ottenuto viene restituito dalla funzione.

Se la funzione è stata dichiarata di tipo `T` diverso da `void`, e l'espressione è di tipo $\texttt{T}' \neq \texttt{T}$ il valore dell'espressione viene convertito automaticamente, se possibile, al tipo `T` prima di restituirlo come valore della funzione. Ad esempio in una funzione di tipo `int` l'istruzione

```
return 2.3;
```

è equivalente a

```
return (int) 2.3;
```

che è lo stesso di

```
return 2;
```

Se dopo l'istruzione `return` non vi è nessuna espressione la funzione restituisce nessun valore e quindi la funzione deve essere di tipo `void`. Lo Standard C89 permette di utilizzare un'istruzione `return` senza nessuna espressione anche in funzioni non di tipo `void`, tuttavia in questo caso se si richiede il valore della funzione questo non è definito. Nel C99 utilizzare un'istruzione `return` senza nessuna espressione in una funzione non di tipo `void` non è permesso.

Nello Standard C se una funzione è stata dichiarata di tipo `void` è un errore fornire un'espressione dopo l'istruzione `return`.

Se l'esecuzione della funzione raggiunge la fine del corpo della funzione senza aver incontrato un'istruzione `return` l'esecuzione della funzione termina come se fosse stata eseguita un'istruzione `return` senza nessuna espressione, per cui se la funzione è stata definita di tipo diverso da `void` il risultato è indeterminato.

La morale di questa storia è che bisogna sempre mettere nel corpo di una funzione almeno un'istruzione `return`, che ne termini l'esecuzione, seguita da un'espressione compatibile con

il tipo della funzione. Eventuali conflitti tra il tipo della funzione ed il valore restituito attraverso l'istruzione `return` possono essere segnalati dal compilatore.

Come esempio di definizione di funzione consideriamo la definizione di una funzione che determina il valore minore tra due valori dati, anche se spesso questo viene fatto utilizzando una espressione condizionata. e una macro.

```
/*************************************
 * Function: minimum(x,y)            *
 *           minore tra x e y        *
 *************************************/
double minimum(double x, double y)
{
  double min;

  min = x;
  if ( y < min) min = y;
  return min;
}
```

La definizione del tipo funzione inizia con la dichiarazione

```
double minimum(double x, double y)
```

che dichiara che l'identificatore `minimum` è associato ad una funzione che prende due parametri, `x` e `y`, entrambi di tipo `double` e restituisce un valore di tipo `double`.

Le istruzioni racchiuse dalle parentesi graffe

```
{
  double min;

  min = x;
  if ( y < min) min = y;
  return min;
}
```

sono il corpo della funzione e vengono eseguite ogni volta che la funzione viene chiamata. In questo caso sono le istruzioni per determinare il valore minore tra due. Il corpo è terminato dall'istruzione `return` che interrompe l'esecuzione della funzione e restituisce come valore della funzione il valore della variabile `min`.

L'istruzione `return` può comparire più volte, per cui avremmo anche potuto scrivere la funzione come

```
/*************************************
 * Function: minimum(x,y)            *
 *           minore tra x e y        *
 *************************************/
double minimum(double x, double y)
{
  if ( x < y) return x;
```

```
    return y;
}
```

In questo caso a seconda dei valori di `x` e `y` viene eseguita una o l'altra delle due istruzioni `return`. In entrambi i casi l'esecuzione della funzione viene interrotta e la funzione restituisce un valore che dipende da quale delle due istruzioni `return` è stata eseguita.

2.20.2. Dichiarazione del tipo funzione

Come tutti gli oggetti in C anche le funzioni devono essere dichiarate prima di del loro utilizzo. Nel caso del tipo funzione nella dichiarazione bisogna fornire oltre al nome e al tipo della funzione anche il numero ed il tipo dei parametri formali. Una funzione viene quindi dichiarata con il prototipo

```
type (function_name)(type_1 parameter_1, ... , type_n parameter_n);
```

La dichiarazione differisce dalla definizione della funzione per il fatto che il prototipo è seguito dal punto e virgola "`;`" e non dal corpo della funzione. Siccome spesso le dichiarazioni sono fatte "copiando" il prototipo dalla definizione di funzione un errore tipico è la mancanza del punto e virgola "`;`" alla fine.

Nel caso della nostra funzione `minimum()` la dichiarazione è quindi:

```
double minimum(double x, double y);
```

In realtà le informazioni realmente necessarie sulla funzione sono, oltre al suo nome e tipo, il numero ed il tipo dei parametri ma non il loro nome. I parametri delle funzioni sono infatti dei parametri formali il cui utilizzo è limitato al corpo della funzione per cui il loro nome è rilevante solo nella definizione del tipo funzione ma non nella dichiarazione. Di conseguenza il nome dei parametri formali può essere omesso nel prototipo della funzione quando questo compare in una dichiarazione. Le due dichiarazioni di funzione

```
type function_name(type_1 parameter_1, ... , type_n parameter_n);
type function_name(type_1, ... , type_n);
```

sono quindi perfettamente equivalenti. Ad esempio per la nostra funzione `minimum()` si può utilizzare anche la dichiarazione

```
double minimum(double, double);
```

L'aggiunta dei nomi dei parametri formali, anche se non necessari, tuttavia aumenta in genere la chiarezza del programma.

Se la funzione viene utilizzata dopo che sia stata data la sua definizione la dichiarazione non è ovviamente necessaria e può essere omessa.

Funzioni con un numero variabile di parametri

Il linguaggio C ammette funzioni con un numero variabile di parametri o parametri di tipo variabile. Un esempio è la funzione `fprintf()`. La dichiarazione di una funzione con un numero variabile di parametri, o parametri di tipo variabile, viene fatta indicando nel prototipo

della funzione con "`,...`" (una virgola e tre punti) la parte variabile della lista dei parametri. Ad esempio la dichiarazione della funzione `fprintf()` è

```
int fprintf(FILE *stream, const char *format, ...);
```

La definizione di una funzione con un numero variabile di parametri, o parametri di tipo variabile, è un pò più delicata perché il modo con cui i valori degli argomenti in una chiamata di funzione sono assegnati ai parametri della funzione dipende dal sistema. Di conseguenza per una maggiore portabilità dei programmi queste funzioni vengono definite nello Standard C utilizzando le *utilities* fornite dal file di header `stdarg.h`. Se si usano queste utilities la funzione deve necessariamente avere almeno un parametro di tipo fissato prima della parte variabile. La definizione delle funzioni con un numero variabile di parametri sarà discussa più avanti, una volta introdotti gli strumenti necessari.

2.20.3. Uso del tipo funzione: chiamata a funzione

Le funzioni vengono utilizzate effettuando una chiamata a funzione scrivendo l'identificatore della funzione seguito da una lista di espressioni racchiuse dalle parentesi "`(`", "`)`"

```
function_name(expression_1, ..., expression_n)
```

Le espressioni `expression_1`, ..., `expression_n` sono chiamate gli argomenti della funzione, ed il loro valore viene assegnato nella chiamata ai corrispondenti parametri formali della funzione. Le parentesi vanno messe poiché l'identificatore della funzione viene interpretato come "chiamata a funzione" solo se seguito dalle parentesi "`(`", "`)`".

Quando si incontra una chiamata a funzione le espressioni tra le parentesi tonde vengono valutate ed il valore assegnato al parametro formale corrispondente nella definizione della funzione: il valore della prima espressione al primo parametro, il valore della seconda espressione al secondo parametro e così via. Se necessario prima dell'assegnazione viene effettuata una conversione implicita, o un cast esplicito se indicato, dal tipo del valore dell'espressione al tipo del parametro formale. Il programma continua con l'esecuzione del corpo della funzione fino a quando non viene incontrata un'istruzione `return` che termina la funzione restituendo il controllo del programma all'istruzione che ha chiamato la funzione. L'esecuzione del programma riprende quindi da questa istruzione utilizzando, se richiesto dal contesto, il valore restituito dalla funzione.

È evidente che il carattere '`,`' viene utilizzato nella lista degli argomenti della chiamata come separatore e non come operatore di una *comma expression*, cosicché l'espressione

```
f(a, b+c, d*e)
```

viene sempre interpretata come chiamata a funzione e non come espressione sequenziale. Quello che è meno evidente è che mentre quando '`,`' figura come operatore le espressioni sono valutate nell'ordine da sinistra a destra, quando '`,`' figura come separatore l'ordine con cui sono valutate le espressioni non è fissato. Il risultato è che l'ordine con cui sono valutate le espressioni che figurano come argomenti in una chiamata di funzione non è stabilito e può dipendere dal sistema.

Siamo ora in grado di scrivere un semplice programma che utilizza la funzione `minimum()`

Programma: `test-func_min.c`

```c
#include <stdio.h>

double minimum(double x, double y);

int main(void)
{
  int    i, j;
  double x, y;

  printf("min(%g,%g) = %g\n", 1.0, 2.0, minimum(1.0, 2.0));
  x = 4.0;
  y = 3.0;
  printf("min(%g,%g) = %g\n", x, y, minimum(x, y));
  i = 7;
  j = 8;
  printf("min(%g,%g) = %g\n", (double) i, (double) j,
                              minimum((double) i, (double) j));

  return 0;
}

/*************************************
 * Function: minimum(x,y)            *
 *           minore tra x e y        *
 *************************************/
double minimum(double x, double y)
{
  if ( x < y) return x;
  return y;
}
```

Osserviamo che quando la funzione `minimum()` viene chiamata con le variabili `i` e `j` di tipo `int` viene effettuato un cast esplicito al tipo `double`. In realtà in questo caso il cast non sarebbe stato necessario poiché la conversione viene effettuata automaticamente prima di assegnare il valore al corrispondente parametro formale della funzione. Un buon stile di programmazione suggerisce tuttavia di utilizzare casts espliciti ogni qual volta il tipo degli argomenti nella chiamata della funzione sia diverso dal tipo richiesto dalla definizione della funzione. I due casts negli argomenti della funzione `printf()` sono invece necessari per convertire il valore nel tipo appropriato alla direttiva di conversione floating-point "`%g`".

In questo esempio si è usato un prototipo con il nome dei parametri formali, ma si sarebbe potuto usare

```c
double minimum(double, double);
```

ovvero, dal momento che non vi è nessuna relazione tra i nomi dei parametri formali utilizzati nella definizione della funzione e quelli eventualmente utilizzati nella dichiarazione (possono essere omessi dal prototipo!) si sarebbe potuto benissimo utilizzare

```c
double minimum(double value_1, double value_2);
```

o qualsiasi altro nome.

La dichiarazione non sarebbe stata necessaria se la definizione della funzione `minimum()` fosse stata messa nel file sorgente prima della funzione che la utilizza, la funzione `main()` in questo caso:

Programma: `test-func_min2.c`

```
#include <stdio.h>

/*************************************
 * Function: minimum(x,y)            *
 *           minore tra x e y        *
 *************************************/
double minimum(double x, double y)
{
  if ( x < y) return x;
  return y;
}

int main(void)
{
  int    i, j;
  double x, y;

  printf("min(%g,%g) = %g\n", 1.0, 2.0, minimum(1.0, 2.0));
  x = 4.0;
  y = 3.0;
  printf("min(%g,%g) = %g\n", x, y, minimum(x, y));
  i = 7;
  j = 8;
  printf("min(%g,%g) = %g\n", (double) i, (double) j,
                              minimum((double) i, (double) j));

  return 0;
}
```

Una struttura di programmazione di questo tipo è tuttavia sconsigliata perché obbliga a scorrere tutto il file sorgente prima di trovare la funzione principale `main()`.

2.20.4. Passaggio per valore e passaggio per indirizzo

Il linguaggio C utilizza per il passaggio dei valori ai parametri delle funzioni il passaggio per valore. Questo vuol dire che quando una funzione viene chiamata ai parametri formali della funzione vengono assegnati i valori delle espressioni che figurano nella lista degli argomenti specificati tra le parentesi "`(`", "`)`". Di conseguenza la chiamata

```
function_name(expression_1, ... ,expression_n)
```

viene interpretata nell'esecuzione della funzione come se nel corpo della funzione vi fossero le dichiarazioni

```
type_1 parameter_1 = expression_1;
type_2 parameter_2 = expression_2;
...
type_n parameter_n = expression_n;
```

I parametri formali della funzione sono quindi delle variabili distinte da eventuali variabili esterne alla funzione che possano comparire nella lista degli argomenti con cui la funzione viene chiamata. Ad esempio nei due programmi precedenti abbiamo utilizzato le istruzioni

```
x = 4.0;
y = 3.0;
printf("min(%g,%g) = %g\n", x, y, minimum(x, y));
```

Le variabili `x` e `y` sono variabili distinte dalle variabili `x` e `y` utilizzate nel corpo della funzione. Quando la funzione `minimum()` viene chiamata il valore delle variabili `x` e `y` della funzione `main()` viene assegnato alle variabili `x` e `y` della funzione `minimum()` ma ciascuna variabile mantiene la propria identità, anche se hanno lo stesso nome.

I parametri formali delle funzioni sono creati, con il valore del rispettivo argomento della chiamata, ogni volta che la funzione viene chiamata e distrutti quando l'esecuzione della funzione termina. Di conseguenza i parametri formali di chiamate differenti della stessa funzione sono variabili distinte. Usualmente i parametri formali vengono creati in una zona di memoria chiamata *stack*.

Il seguente semplice programma illustra il passaggio per valore

Programma: `show-pass.c`

```
#include <stdio.h>

int sum(int n);

int main(void)
{
  int n, somma;

  n = 5;
  printf("1_main > n vale: %d\n", n);
  somma = sum(n);
  printf("2_main > n vale: %d\n", n);
  printf("somma da 1 a %d =  %d\n", n, somma);
  return 0;
}

int sum(int n)
{
  int tmp = 0;

  printf("1_func > n vale: %d\n", n);
  for(; n > 0; --n) tmp += n;
  printf("2_func > n vale: %d\n", n);
  return tmp;
```

```
}
```

Quando il programma viene eseguito si ha

```
1_main > n vale: 5
1_func > n vale: 5
2_func > n vale: 0
2_main > n vale: 5
somma da 1 a 5 =  15
```

da cui si vede chiaramente che la variabile `n` della funzione `main()` è distinta dalla variabile `n` della funzione `sum()`.

La conseguenza più evidente del passaggio per valore è che non è possibile modificare dall'interno della funzione il valore delle variabili esterne passate come argomenti della funzione. Ad esempio la seguente funzione

```
void swap(double x, double y)   /* Errata */
{
  double temp;
  temp = x;
  x    = y;
  y    = temp;
  return;
}
```

non scambia i valori delle variabili `x` e `y` con cui viene chiamata, ma solo il valore delle variabili `x` e `y` della funzione. Per modificare il valore di una variabile dall'interno di una funzione bisogna passare esplicitamente alla funzione l'indirizzo di memoria della variabile di cui si vuole modificare il valore utilizzando un puntatore. Abbiamo già incontrato questo problema con le funzioni di Input che necessariamente devono modificare il valore delle variabili. Riprenderemo questo punto una volta introdotte le variabili di memoria ed i puntatori.

L'alternativa al passaggio per valore è il passaggio per indirizzo, utilizzato ad esempio dal linguaggio di programmazione FORTRAN. Nel passaggio per indirizzo, come indica il nome stesso, viene passato alla funzione l'indirizzo di memoria delle variabili che compaiono come argomenti nella chiamata della funzione e non il loro valore. I questo caso i parametri formali della funzione sono a tutti gli effetti equivalenti alla variabili esterne passate alla funzione in quanto entrambi sono associati alle stesse zone di memoria. Di conseguenza se ad esempio il valore di una variabile passata ad una funzione, come nell'esempio della funzione `swap()`, viene cambiato all'interno della funzione questo risulterà cambiato anche al di fuori della funzione.

2.20.5. Funzioni senza parametri

Il linguaggio C permette di definire funzioni che non prendono parametri formali. In questo caso nel prototipo della dichiarazione della funzione la lista dei parametri deve essere sostituita dallo specificatore di tipo `void`

```
type (function_name)(void);
```

dove `type` è il tipo `T` della funzione. Lo specificatore `void` è necessario altrimenti l'istruzione

```
type (function_name)();                /* Errata */
```

verrebbe interpretata come la definizione della funzione senza parametri `function_name()` nello stile tradizionale.

Nel prototipo della definizione di una funzione senza parametri la lista dei parametri può essere semplicemente omessa

```
type function_name()
{
 function_body
}
```

tuttavia è preferibile utilizzare la forma estesa

```
type function_name(void)
{
 function_body
}
```

per indicare esplicitamente che la funzione non prende parametri.

Una funzione senza parametri viene richiamata specificando il suo nome seguito dalle parentesi "`()`":

```
function_name()
```

come mostrato nel seguente frammento di programma.

```
int next_value(void);

int main(void)
{
  int value
  ...
  value = next_value();
  ...
  value = next_value();
  ...
  return 0;
}

int next_value(void)
{
  int value;

  printf("Valore =");
  scanf("%d", &value);
  return value;
}
```

Le parentesi dopo il nome della funzione sono obbligatorie anche nel caso in cui la funzione non prenda nessun parametro, nel qual caso la lista degli argomenti è vuota "`()`", poiché

l'identificatore di una funzione viene interpretato come "chiamata a funzione" solo se seguito dalle parentesi "(","(".

2.20.6. Programmazione Strutturata

L'uso di funzioni permette di costruire il programma per mezzo di ben definite funzioni che assolvono compiti specifici. In questo modo non solo è possibile ridurre la soluzione di problemi complessi ad una serie di problemi più semplici, ma al contempo si ha una visione chiara della struttura logica del programma. L'utilizzo di funzioni quindi non solo semplifica la scrittura di un programma ma anche le eventuali modifiche ed estensioni future. Va da se che una programmazione ben strutturata mediante un uso corretto di funzioni permette di realizzare programmi *modulari* molto flessibili e facilmente adattabili a situazioni diverse.

Ad esempio il programma per la rappresentazione binaria di un numero intero con segno avrebbe potuto essere scritto schematicamente come

```
int main(void)
{
    read_number();
    find_repr();
    print_repr();
    return 0;
}
```

demandando tutti i dettagli per le operazioni di Input/Output e per la rappresentazione binaria del numero a ciascuna funzione. In questo modo non solo la struttura logica del programma è ben chiara ma per passare da un tipo di rappresentazione ad un'altra è sufficiente sostituire le funzioni con le corrispondenti funzioni adatte al tipo di rappresentazione voluta, senza dover quindi riscrivere completamente il programma.

In linea di principio la programmazione strutturata può essere affrontata in due modi differenti ed opposti. Si può utilizzare un approccio *bottom-up* in cui si comincia a scrivere le funzioni di base con cui costruire poi tutto il programma. Oppure si può usare un approccio *top-down* cominciando invece a scrivere la struttura generale del programma lasciando i dettagli nelle funzioni che poi verranno scritte. Non vi sono forti motivazioni per preferire un metodo all'altro. Nella pratica per lo sviluppo di un programma si usano infatti entrambi i metodi a seconda del gusto del programmatore, del problema da risolvere e così via.

2.21. Funzioni ricorsive (Rev. 2.0.2)

Una funzione è detta *ricorsiva* se direttamente o indirettamente richiama se stessa. Nel linguaggio C tutte le funzioni possono essere utilizzate ricorsivamente.

Nella scrittura di funzioni ricorsive si devono seguire due regole basilari:

- Deve esistere una condizione di fine per la ricorrenza;
- Ad ogni chiamata devono semplificare, o per lo meno non complicare, il problema.

L'esempio classico di funzione ricorsiva è la funzione che calcola il fattoriale:

```
int fact(int num)
{
  if (num == 0) return 1;
  return ( num * fact(num - 1) );     /* ricorsiva */
}
```

Questa funzione soddisfa le richieste. Infatti ha una condizione di fine per la ricorrenza: quando `num` raggiunge il valore `0`, ed inoltre semplifica il problema: il calcolo di `fact(num - 1)` è più semplice di quello di `fact(num)`.

La condizione di fine della ricorrenza è fondamentale. Supponiamo infatti di chiamare la funzione `fact()` con un argomento negativo, ad esempio `fact(-3)`. Il calcolo di `fact(-3)` richiede il calcolo di `fact(-4)` che a sua volta richiede quello di `fact(-5)` e così via. In questo caso non esiste il punto finale perchè `num` non potrà mai essere uguale a `0` ed programma abortisce con un messaggio di "*stack overflow*" o simile. Questo è quello che si chiama un errore di ricorrenza infinita.

Molti algoritmi hanno sia una formulazione *ricorsiva* che *iterativa*. Ad esempio la funzione `fact()` può essere scritta in forma iterativa come:

```
int fact(int num)
{
  int prod = 1;

  for ( ; num > 1; --num) prod *= num;    /* iterativa */

  return prod;
}
```

Formulazione ricorsiva o iterativa?

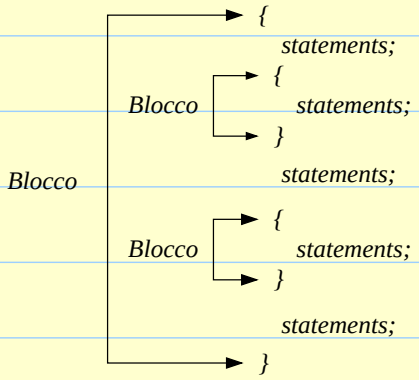
In generale la formulazione ricorsiva risulta piú elegante e richiede meno variabili per effettuare lo stesso calcolo. È la ricorrenza che si occupa di mantenere i valori intermedi su uno "stack". Per contro l'utilizzo dello stack in generale richiede più tempo e memoria per cui un programma che utilizzi funzioni ricorsive può risultare molto meno efficiente di uno che invece utilizzi funzioni iterative.

D'altro canto uno schema ricorsivo è di solito più facile da scrivere, comprendere e mantenere. Quindi la questione della scelta tra uno schema ricorsivo ed uno iterativo si basa molto sul problema da trattare.

In assenza di molte informazioni un modo di procedere potrebbe essere quello di utilizzare uno schema ricorsivo, più facile da scrivere. Se poi il programma dovesse presentare grosse inefficienze provare a sostituire lo schema ricorsivo con l'equivalente iterativo.

2.22. Scopo, Visibilità e Classe (Rev. 2.1.1)

Tutti gli oggetti in C hanno una propria validità "spaziale" e "temporale" determinata dalla loro dichiarazione. Per descrivere queste proprietà in modo appropriato è utile introdurre il concetto di *blocco*. Un blocco è un insieme di definizioni ed istruzioni delimitato da una coppia di parentesi graffe "{}". Ricordiamo che il C89 richiede che le dichiarazioni in un blocco siano poste sempre all'inizio del blocco prima della prima istruzione. Il C99 non è così restrittivo e permette dichiarazioni in qualsiasi parte del blocco. Un blocco può a sua volta contenere altri blocchi:



Un blocco forma usualmente una istruzione composta ma può anche essere il corpo di una funzione o altro, in ogni caso indipendentemente dal suo significato un blocco delimita sempre una parte o regione del programma.

2.22.1. Scopo

Lo scopo di un oggetto è la parte del programma in cui la sua dichiarazione è visibile. Se lo scopo è limitato ad una parte del programma lo scopo è detto genericamente *locale* o *di blocco*, in caso contrario lo scopo è *globale*.

La validità di una dichiarazione a scopo globale si estende dal punto in cui viene fatta fino alla fine del file sorgente del programma per cui l'oggetto dichiarato può essere utilizzato, se visibile, dalla sua dichiarazione fino alla fine del file.[5] Le dichiarazioni fatte fuori da tutti i blocchi hanno sempre scopo globale.[6] Al contrario le dichiarazioni fatte all'interno di un blocco hanno sempre scopo locale e la loro validità si estende dal punto in cui sono fatte fino alla fine del blocco, includendo eventualmente i blocchi in esso contenuti. Un oggetto dichiarato con scopo di blocco può essere utilizzato, se visibile, solamente nel blocco e nei blocchi in esso contenuti.

Il seguente frammento di programma illustra il concetto di scopo di un oggetto.

```
int function(int formal_param);   /* valida ovunque     */
```

[5] Considereremo più avanti il caso di un programma composto da più files sorgente.

[6] Le variabili globali se non inizializzate esplicitamente nella dichiarazione sono inizializzate automaticamente a zero.

```
double global;                      /* valida ovunque     */

int main(void)
{
  double local;                     /* valida in main e   */
                                    /* blocchi contenuti  */
  global = 1.0;
  local  = 2.0;

  {
    double more_local;              /* valida nel blocco  */

    more_local = global * local;
  }
  return 0;
}

int function(int func_param)
{
   int func_local;                  /* valida in function */

   func_local = 4 * func_param;
   printf("global = %f\n", global);
   return (func_local);
}
```

In questo esempio la dichiarazione della variabile `global` è fuori da tutti i blocchi per cui la sua validità si estende dal punto in cui viene effettuata fino alla fine del file sorgente del programma. La variabile `global` è di conseguenza visibile sia nella funzione `main()` che nella funzione `function()`. Il suo scopo è globale.

Lo stesso vale per la dichiarazione della funzione `function()`. Essendo fatta fuori da tutti i blocchi la sua validità si estende dal punto in cui viene fatta fino alla fine del file sorgente del programma. Lo scopo è quindi globale e la funzione `function()` è visibile in tutto il file sorgente.

Al contrario la variabile `local` è dichiarata nel blocco che definisce il corpo della funzione `main()` per cui il suo scopo è locale. La sua dichiarazione è infatti valida solo dal punto in cui viene fatta fino alla fine della funzione. La variabile `local` è quindi visibile solo nel corpo della funzione `main()`, inclusi eventuali blocchi in esso contenuti, ma non al di fuori di esso.

La dichiarazione della variabile `more_local` è all'interno di un blocco contenuto nel corpo della funzione `main()` per cui la sua validità si estende dal punto in cui viene fatta fino alla fine del blocco. Il suo scopo è quindi di blocco e la variabile `more_local` non è visibile al di fuori di esso, ad esempio in altre parti del corpo della funzione `main()`.

La dichiarazione della variabile `func_local` è fatta nel corpo della funzione `function()` e quindi la sua validità si estende dal punto in cui viene fatta fino alla fine della funzione. Il suo scopo è locale e la variabile `func_local` è visibile solo nel corpo della funzione `function()` e non al di fuori di esso.

La dichiarazione del parametro `func_param` viene fatta nel prototipo della definizione della

funzione `function()`, la sua dichiarazione ha quindi validità dal punto in cui viene effettuata fino alla fine della funzione. Lo scopo è quindi locale ed il parametro `func_param` è visibile solo nel corpo della funzione `function()`.

Infine il parametro formale `formal_param` viene dichiarato nel prototipo della dichiarazione della funzione `function()`. La sua validità si estende quindi dal punto in cui viene effettuata fino alla fine del prototipo. Il suo scopo è di nuovo locale, ed il parametro formale `formal_param` è visibile solo nel prototipo. A volte per i parametri formali dichiarati nei prototipi delle dichiarazioni di funzione si usa il termine *scopo di prototipo* per distinguerli dai parametri dichiarati nei prototipi delle definizioni di funzione la cui visibilità si estende invece a tutto il corpo della funzione.

In conclusione in questo esempio nel blocco più interno della funzione `main()` possono essere utilizzate le variabili `global`, `local` e `more_local` e la funzione `function()` ma non la variabile `func_local`. Nel corpo della funzione `main()`, ma fuori del blocco più interno, possono essere utilizzate solo le variabili `global` e `local` e la funzione `function()` ma non le variabili `more_local` e `func_local`. Infine nel corpo della funzione `function()` possono essere usate solo le variabili `func_param`, `global` e `func_local`, e la funzione stessa `function()`, ma non le variabili `local` e `more_local`. Infine il parametro formale `formal_param` può essere usato solo nel prototipo della dichiarazione di funzione, per l'appunto come parametro formale.

2.22.2. Visibilità

Per poter utilizzare un oggetto in una parte del programma non è sufficiente che il suo scopo lo permetta è anche necessario che la sua dichiarazione sia visibile in quella parte di programma. La dichiarazione di un oggetto è visibile in un certo contesto se l'identificatore o nome dell'oggetto è associato alla dichiarazione dell'oggetto, ossia se l'identificatore si riferisce all'oggetto in questione. Questa affermazione che a prima vista può sembrare banale in realtà non lo è. Il motivo è che quando un oggetto viene dichiarato gli viene assegnata una zona di memoria identificata tramite l'identificatore specificato nella dichiarazione, per cui se lo stesso identificatore viene utilizzato più volte in dichiarazioni successive l'identificatore sarà associato di volta in volta ad oggetti fisicamente differenti. Questi potranno differire non solo per zona di memoria ma anche per tipo. Di conseguenza può capitare che in una parte del programma un oggetto sia nascosto (*hidden*) dalla dichiarazione di un altro oggetto che utilizza lo stesso identificatore ed il cui scopo si sovrappone quindi a quello dell'oggetto in questione. In altre parole l'oggetto originale non è più visibile e non può quindi essere utilizzato nonostante il suo scopo lo permetta perché l'identificatore non è associato alla sua dichiarazione ma ad un'altra.

Il seguente programma illustra il concetto di visibilità di un oggetto.

```
1  #include <stdio.h>
2
3  int total;
4
5  int main(void)
6  {
7    int count;            /* (1) count locale a main */
```

```
 8
 9    count = 0;
10    total = 0;
11
12    {
13      int count;          /* (2) count locale al blocco      */
14                          /*     nasconde la variabile count */
15      count = 0;          /*     definita in main            */
16
17      while (count < 10) {
18        total += count;
19        count++;
20      }
21
22      printf("blocco -> count vale: %d\n", count);
23    }
24
25    count++;
26    printf("main   -> count vale: %d\n", count);
27
28    return 0;
29  }
```

Quando il programma viene eseguito il risultato è

```
blocco -> count vale: 10
main   -> count vale: 1
```

Dall'analisi del programma si vede che la variabile `count` definita alla linea 7 con scopo il corpo della funzione `main()` viene nascosta dalla definizione alla linea 13 che utilizza lo stesso identificatore `count` per dichiarate una variabile con scopo locale al blocco. Infatti sebbene lo scopo della dichiarazione alla linea 7 permetta di utilizzare la variabile all'interno del blocco, la sua dichiarazione ha validità dal punto in cui viene effettuata fino alla fine della funzione `main()`, la successiva dichiarazione alla linea 13 nasconde la variabile più esterna associando l'identificatore `count` alla nuova variabile con scopo limitato al blocco più interno. La visibilità della variabile `count` definita alla linea 7 è quindi ristretta a tutto il corpo della funzione `main()` con l'esclusione del blocco più interno anche se la sua dichiarazione avrebbe in principio valore in tutto il corpo della funzione, blocco incluso.

È possibile ripetere questa operazione creando una gerarchia di oggetti ciascuno dei quali nasconde tutti i precedenti. Sebbene questo sia possibile è da evitare, a meno che non si sappia bene cosa si stia facendo, poiché altrimenti potrebbe diventare difficile sapere a quale oggetto ci si stia riferendo in un certo contesto, con risultati facilmente prevedibili.

Come detto più volte la validità di una dichiarazione inizia nel punto in cui la dichiarazione viene effettuata. Questa affermazione abbastanza ovvia in congiunzione con la possibilità di nascondere le definizione di un oggetto può dare origini a situazioni come la seguente:

```
...
{
  int x;
```

```
    x = 2;

    {
      int y = x;
      int x = 1;
      ...
    }
}
```

Che valore viene assegnato alla variabile y? Siccome lo Standard ISO C richiede una definizione abbia validità a partire dal punto in cui viene effettuata alla variabile y viene assegnato il valore della varibile x esterna, ossia il valore 2. La variabile viene poi nascosta nel blocco dalla nuova definizione della variabile x. Tuttavia alcuni compilatori non Standard ISO C potrebbero pensarla diversamente. Si consiglia quindi di non utilizzare mai costruzioni di questo tipo, che oltre a tutto sono considerate un pessimo stile di programmazione.

Overloading

Il linguaggio C permette di utilizzare contemporaneamente lo stesso identificatore per oggetti differenti. In questo caso l'oggetto specifico a cui si riferisce l'identificatore viene determinato dal contesto in cui l'identificatore viene utilizzato. Ad esempio uno stesso identificatore può essere usato sia per una variabile che per una label dell'istruzione `goto`. Quando l'identificatore viene utilizzato in una espressione verrà associato alla variabile, se invece viene utilizzato in una istruzione `goto` sarà associato con una label. Se un identificatore viene associato contemporaneamente a più oggetti si dice che è stato "*overloaded*".

Chiaramente non è possibile utilizzare contemporaneamente nello stesso blocco o fuori da ogni blocco lo stesso identificatore per oggetti dello stesso tipo perché le dichiarazioni sarebbero in *conflitto*.

In genere per poter fare un *overloading* dell'identificatore non è sufficiente che gli oggetti siano di tipo differente, ad esempio non è possibile utilizzare contemporaneamente lo stesso identificatore per una funzione ed una variabile. Il linguaggio C divide i vari oggetti in classi chiamate *classi di overloading* o *spazio dei nomi*. È possibile fare un *overloading* dell'identificatore solo se gli oggetti appartengono a due classi di *overloading* differenti. La classi utilizzate dal linguaggio C sono

Classe	Identificatori
Macros	Nomi dei parametri delle macros del preprocessore.
Labels	Identificatori che seguono l'istruzione `goto` o sono seguiti da ":".
Strutture, unioni e tipo enumerativo	Identificatori del tipo struttura, tipo unione e tipo enumerativo. Questi identificatori seguono sempre le keywords `struct`, `union` e `enum`.

Classe	Identificatori
Membri strutture e unioni	Identificatori utilizzati per individuare i membri o campi di una struttura od una unione.
Altri nomi	Tutto quello che non appartiene alle precedenti classi. A questa classe appartengono in particolare gli identificatori delle variabili e delle funzioni, `typedef` *names* e i nomi delle costanti enumeritive.

Per il momento noi abbiamo visto solo identificatori appartenenti alle prime due classi e all'ultima. I tipi strutture, unione ed enumerativo ed i `typedef` *names* saranno visti più avanti e sono stati inclusi solo per completezza.

È chiaro che una dichiarazione può nasconderne un'altra solo se entrambi si riferiscono ad oggetti nella stessa classe di *overloading*, in altre parole quando un identificaore viene *overloaded* ogni associazione ha un suo scopo.

2.22.3. Classe di memorizzazione

Un oggetto può avere una validità temporale sia permanente che temporanea. Gli oggetti a validità permanente sono anche detti *statici* (*static*) mentre quelli a validità temporanea *automatici* (*automatic*) in riferimento ai due diversi metodi di memorizzazione utilizzati.

Gli oggetti statici vengono creati, ed eventualmente inizializzati, prima che inizi l'esecuzione del programma e rimangono "in vita" fino alla fine di esso. Un oggetto con scopo globale è sempre statico.

Gli oggetti automatici sono invece creati, ed eventualmente inizializzati, durante l'esecuzione del programma quando vengono dichiarati e rimangono "in vita" per la durata del loro scopo. Tutte le dichiarazioni con scopo locale, se non specificato diversamente, sono automatiche. Questo vuol dire, ad esempio, che le variabili dichiarate in un blocco sono create ed inizializzate all'inizio del blocco ogni volta che l'esecuzione del programma entra nel blocco e distrutte quando se ne esce.

Quando le variabili in classe di memorizzazione automatica vengono dichiarate gli viene assegnato uno spazio di memoria ma non un valore, in altre parole se una variabile automatica non viene inizializzata esplicitamente il suo valore non è definito. Al contrario una variabile in classe di memorizzazione statica viene sempre inizializzata a zero in assenza di una inizializzazione esplicita nella sua dichiarazione. È importante ricordarsi che una variabile statica viene inizializzata con il valore specificato nella dichiarazione, o con il valore zero se questo non viene dato, una sola volta quando viene dichiarata.

Le variabili automatiche sono create in una zona di memoria chiamato *stack*. Quando una variabile automatica viene distrutta lo spazio liberato viene restituito allo stack e riutilizzato. Se si creano troppe variabili automatiche contemporaneamente si può superare la capacità dello stack, ed in questo caso si ha un errore di "Stack overflow". Le dimensioni dello stack dipendono dal sistema e dal compilatore. Su molti sistemi UNIX il programma automatica-

mente prende tutto lo spazio disponibile per lo stack, su altri sistemi le dimensioni dello stack possono essere cambiate mediante parametri del compilatore. Se si hanno problemi di stack conviene usare variabili statiche.

La validità temporale di un oggetto è determinata sulla base alla sua classe di memorizzazione, o semplicemente classe, specificata nella sua dichiarazione con uno dei seguenti *specificatori*:

Specificatori di classe:

```
auto  extern  register  static
```

La possibilità di utilizzare uno specificatore di classe in genere dipende sia dall'oggetto che da contesto della dichiarazione. Ad esempio nella dichiarazione dei parametri formali nei prototipi delle funzioni può essere utilizzato solo lo specificatore `register`. Ogni dichiarazione può contenere, se permesso, al massimo un solo specificatore di classe che di solito viene messo al primo posto della dichiarazione. Sebbene questo non sia necessario questa è la disposizione suggerita dallo standard Standard C.

Classe `auto`

Lo specificatore di classe di memorizzazione `auto` può essere utilizzato solo nelle dichiarazioni di variabili all'interno di un blocco per indicare che la variabile è automatica. La classe `auto` è la classe di memorizzazione di *default* per le variabili a scopo locale per cui lo specificatore `auto` viene usato molto raramente.

Classe `register`

Nello Standard C lo specificatore di classe di memorizzazione `register` può essere usato sia nelle dichiarazioni di variabili con scopo locale che nelle dichiarazioni dei parametri nei prototipi delle funzioni. Una variabile di classe di memorizzazione `register` è automatica ma, a differenza delle variabili di classe memorizzazione `auto`, viene memorizzata su registri di memoria ad accesso veloce. Siccome le variabili di classe `register` sono memorizzate su registri particolari lo Standard C non permette di calcolarne l'indirizzo di memoria.

La classe di memorizzazione `register` viene usata per migliorare le prestazioni dei programmi, tuttavia è bene ricordare che un uso eccessivo di variabili di classe `register` può risultare non solo irrilevante ma addirittura controproducente. Inoltre con compilatori altamente ottimizzati l'uso della classe di memorizzazione `register` può risultare poco efficace perché questi già utilizzano quando necessario registri di memoria veloce. Su molti compilatori in effetti la classe di memorizzazione `register` viene trattata alla stessa stregua della classe `auto`.

Classe `static`

Lo specificatore di classe `static` può essere usato sia con variabili che con funzioni.

Se `static` viene usato con variabili a scopo locale, dette anche variabili locali, indica che la variabile ha validità temporale statica. Una variabile locale di classe `static` viene creata ed

inizializzata con il valore fornito nella dichiarazione, o con zero se questo non viene dato, una sola volta all'inizio dell'esecuzione del programma. Inoltre, siccome la variabile non viene distrutta quando si esce dal blocco che ne delimita lo scopo, una variabile locale statica mantiene il suo valore da un'esecuzione del blocco che delimita il suo scopo alla successiva, come mostrato dal seguente programma.

Programma: `var_auto-stat.c`

```
#include <stdio.h>

int main(void)
{
  int count;

  for (count = 0; count < 3; ++count)
    {
      int         temp = 1;     /* temporanea */
      static int perm = 1;      /* permanente */
      printf("temp -> %d; perm -> %d\n", temp, perm);
      ++temp;
      ++perm;
    }
  return 0;
}
```

La variabile `temp` è una variabile automatica, mancando infatti nella dichiarazione lo specificatore di classe di memorizzazione viene usata la classe di default `auto`. La variabile `perm` è invece statica essendo dichiarata con lo specificatore `static`. Le variabili `temp` e `perm` hanno entrambe scopo locale al blocco che costituisce il corpo del ciclo `for`, di conseguenza mentre `temp` viene creata e distrutta ad ogni iterazione del ciclo la variabile `perm` mantiene il suo valore da una iterazione all'altra, come mostrato dall'output del programma

```
temp -> 1; perm -> 1
temp -> 1; perm -> 2
temp -> 1; perm -> 3
```

Lo specificatore di classe di memorizzazione `static` usato nella definizione delle funzioni e nella dichiarazione di variabili con scopo globale indica che lo scopo della variabile o della funzione è il file sorgente. In altre parole il nome della variabile globale o della funzione non viene reso noto al *linker* per cui se il programma è composto da più files sorgente qualsiasi riferimento ad esse dagli altri files che compongono il programma sono vietati.

Nella dichiarazione delle funzioni lo specificatore `static` indica che la funzione verrà definita in classe `static` più avanti nel file sorgente.

In conclusione lo specificatore `static` indica non solo che l'oggetto è statico ma anche che lo scopo dell'oggetto non può estendersi oltre al file. La possibilità di creare delle variabili, funzioni o altri oggetti "privati" ad un file sorgente, ossia utilizzabili sono nel file, risulta particolarmente utile nella scrittura di moduli e librerie.

Classe `extern`

Lo specificatore di classe `extern` può essere usato sia con funzioni che con variabili. Tutti gli oggetti in classe `extern` hanno validità temporale statica.

Lo specificatore `extern` viene utilizzato con scopi opposti a quelli dello specificatore `static` infatti questo specificatore permette di creare oggetti che possono essere utilizzati per scambiare informazioni tra i diversi files sorgente che compongono un programma L'uso congiunto delle classi `static` e `extern` permette di sviluppare programmi molto flessibili strutturati su moduli indipendenti che comunicano solo attraverso oggetti di classe `extern` ed è quindi alla base dello sviluppo, ad esempio, di librerie.

Lo specificatore di classe di memorizzazione `extern` usato nella dichiarazione delle funzioni, sia al di fuori di tutti i blocchi che all'interno di un blocco, indica che la definizione della funzione non si trova necessariamente nello stesso file. Questo è ad esempio il caso di tutte le funzioni delle librerie standard come `printf()`, `pow()` e così via la cui definizione si trova in un file diverso da quello dei programmi che le usano.

Usato nella definizione delle funzioni lo specificatore di classe `extern` indica che la funzione non è locale al file e che quindi è possibile fare dichiarazioni che facciano *riferimento* alla funzione anche da altri files. Tutte le funzioni delle librerie che devono essere visibili fuori dalla libreria sono ovviamente definite con la classe `extern`.

Se nella dichiarazione di funzione non compare lo specificatore di classe di memorizzazione viene assunta per default la classe `extern`. Nel caso della definizione di funzione la classe di memorizzazione di default è `static` se la definizione è preceduta nel file sorgente da una dichiarazione della funzione in classe `static`, altrimenti è `extern`.

Nello Standard C89 se viene effettuata una chiamata a funzione senza che la funzione sia stata prima dichiarata il compilatore assume che la funzione sia stata dichiarata implicitamente in classe `extern` nel blocco più interno che contiene la chiamata. In altre parole se il compilatore trova un identificatore `ident` seguito da una parentesi "`(`" senza che `ident` sia stato precedentemente dichiarato come identificatore di una funzione il compilatore assume che `ident` sia stato dichiarato come

```
extern int ident();
```

nel blocco più interno che contiene la chiamata. Il C99 produce un messaggio di avvertimento e, a seconda del sistema, può interrompere la compilazione o comportarsi come il C89.

Lo specificatore `extern` può essere usato anche nella dichiarazione delle variabili sia al di fuori di tutti i blocchi che all'interno di un blocco ed indica che la dichiarazione è una *referenza* alla definizione della variabile statica fatta altrove nel file o in un altro file. La differenza tra una definizione ed una referenza verrà discussa più in dettaglio nel prossimo paragrafo. Se le variabili globali non sono dichiarate esplicitamente in classe di memorizzazione `static` o `extern` il compilatore assume per *default* la classe `extern`.

L'utilizzo della classe di memorizzazione `extern` permette creare variabili "esterne" che possono essere utilizzate per scambiare informazioni tra files diversi come le variabili globali possono essere usate per scambiare informazioni tra blocchi diversi nello stesso file. I seguenti due files sorgente illustrano questo punto

File: `var_extr1.c`

```
#include <stdio.h>

int x = 1;
int y = 2;
extern int func(void);

int main(void)
{
  printf("main: x -> %3d   y -> %3d\n", x, y);
  printf("main: func          -> %3d\n", func());
  printf("main: x -> %3d   y -> %3d\n", x, y);
  return 0;
}
```

File: `var_extr2.c`

```
#include <stdio.h>

extern int func(void)
{
  extern int x;
  int        y;

  x = 10;
  y = 20;
  printf("funzione:        y -> %3d\n", y);
  {
    extern int y;
    printf("blocco funzione: y -> %3d\n", y);
  }
  return (x + y);
}
```

Nel file sorgente `var_extr1.c` vengono dichiarate le due variabili globali `x` e `y`, che mancando lo specificatore sono in classe di memorizzazione `extern`, e la funzione `func()` dichiarata esplicitamente con classe di memorizzazione `extern` poiché la sua definizione si trova nel file `var_extr2.c`. Lo specificatore della classe di memorizzazione non sarebbe necessario poiché se questo non viene specificato il compilatore assume per default la classe `extern`. Nel file `var_extr2.c` la funzione viene definita con classe di memorizzazione `extern` perché deve essere possibile farvi riferimento dal file `var_extr1.c`. Di nuovo non sarebbe stato necessario specificare la classe esplicitamente.

La funzione `func()` definita in `var_extr2.c` utilizza la variabile globale `x` definita nel file `var_extr1.c` per cui la variabile viene dichiarata come

```
extern int x;
```

Questa non è una definizione poiché la variabile non viene creata ma una referenza alla variabile `x` dichiarata nel file `var_extr1.c`. La dichiarazione è necessaria anche se non viene

creata nessuna variabile per informare il compilatore che l'identificatore si riferisce ad una variabile statica di tipo `int` definita in un altro file. In questo modo la variabile `x` può essere utilizzata sia nella funzione `main()` che nella funzione `func()` anche se sono definite in files diversi. Lo scopo della variabile `x` è tutto il file `var_extr1.c` ed il corpo della funzione `func()`.

La funzione `func()` utilizza due variabili chiamate `y`. La prima introdotta dalla dichiarazione

```
int y;
```

all'inizio del blocco che definisce il corpo della funzione. Questa è una definizione e crea una nuova variabile di tipo `int` con scopo locale al corpo della funzione. La variabile ha lo stesso nome della variabile globale definita nel file `var_extr1.c` ma a parte ciò sono variabili distinte. La seconda variabile `y` utilizzata nel blocco più interno del corpo della funzione è la variabile globale `y` definita nel file `var_extr1.c` ed è dichiarata con la referenza

```
extern int y;
```

In questo caso, diversamente dalla variabile `x`, il suo scopo è solo il blocco e non tutto il corpo della funzione `func()`. Non vi sono conflitti perché la dichiarazione della variabile globale `y` nasconde all'interno del blocco la variabile `y` con scopo locale alla funzione.

Per generare un file eseguibile i due file sorgente vanno compilati separatamente e poi uniti insieme con il linker. Questo si può ottenere facilmente con il comando

```
$ cc var_extr1.c var_extr2.c
```

Quando il programma viene eseguito si ha

```
main: x ->   1   y ->   2
funzione:        y ->  20
blocco funzione: y ->   2
main: func         ->  30
main: x ->  10   y ->   2
```

da cui si vede chiaramente che la variabili `x` e `y` utilizzate dalla funzioni `main()` e nel blocco più interno del corpo della funzione `func()` sono le stesse, mentre la variabile `y` utilizzata nel corpo della funzione `func()` fuori del blocco più interno è una variabile diversa.

Questo esempio mostra inoltre com scambiare informazioni tra funzioni, anche se definite in file sorgente diversi, attraverso variabili globali. Tuttavia il metodo preferito resta, con alcune eccezioni, quello attraverso i parametri delle funzioni ed il valore restituito dalle funzioni poichè questo migliora la modularità del programma e soprattutto riduce la possibilità di risultati indesiderati. È bene ricordare infatti che una variabile globale può essere vista ovunque e quindi anche modificata ovunque, nel bene e nel male.

Definizione e Referenza

Nel linguaggio C a volte la distinzione tra *definizione* e *referenza* è piuttosto confusa soprattutto con le variabili globali. Infatti se con le funzioni è abbastanza chiaro che la dichiarazione è una referenza alla definizione della funzione questa differenza diventa molto meno chiara nel caso di variabili. Ad esempio mentre è evidente che l'istruzione

```
int x = 0;
```

fuori da ogni blocco è una *definizione*: la variabile `x` non esisteva per cui viene definita creandola ed assegnandogli un valore, cosa sia l'istruzione

```
int x;
```

fuori da tutti i blocchi è molto meno chiaro. Questa può essere interpretata sia come una definizione senza inizializzazione che come una referenza con `extern` implicito alla variabile `x` definita altrove, esattamente come una dichiarazione di funzione è una referenza alla definizione della funzione fatta altrove. Per risolvere questa ambiguità i compilatori usano modelli interpretativi che possono variare da un compilatore ad un altro. La tavola seguente riassume il modello adottato dallo Standard C:

Dichiarazione fuori dai blocchi	**Interpretazione**
`int x;`	Referenza se nel file vi è una definizione successiva di `x`.
`int x;`	Definizione se nel file non vi è una definizione successiva di `x`.
`int x = 0;`	Definizione
`extern int x;`	Referenza
`extern int x = 0;`	Definizione

Ad esempio la dichiarazione

```
int x = 1;
```

nel file `var_extr1.c` è una definizione. Analogamente se si fosse utilizzata la forma equivalente

```
int x;
...
int main(void)
{
  x = 1;
  ...
}
```

la dichiarazione `int x;` sarebbe stata in questo caso una definizione poichè il file `var_extr1.c` non contiene successive definizioni della variabile `x`, ma solo un'assegnazione (inizializzazione in questo caso).

In alcuni modelli interpertativi una dichiarazione della forma

```
extern int x = 0;
```

non è considerata valida, per cui alcuni compilatori possono produrre un messaggio di *warning* in fase di compilazione. Per ridurre al minimo i possibili problemi legati all'utilizzo di modelli interpretativi differenti da parte dei compilatori si raccomanda di attenersi alla seguente semplice regola:

1. Utilizzare una sola definizione per ciascuna variabile esterna. Nella dichiarazione che definisce la variabile non includere lo specificatore di classe di memorizzazione `extern` e fornire sempre un valore iniziale, eventualmente arbitrario, come fatto ad esempio nel file `var_extr1.c`:

```
#include <stdio.h>

int x = 1;
...
```

2. Nei files sorgente che non contengono la definizione della variabile esterna fare riferimento alla variabile esterna specificando la classe di memorizzazione `extern` e non fornire un valore iniziale, come fatto nel file `var_extr2.c`:

```
extern int func(void)
{
  extern int x;
  ...
}
```

Infine è bene tener presente che il compilatore C non può controllare che le dichiarazioni di variabili e funzioni esterne fatte in files sorgente differenti siano tutte consistenti fra di loro. È quindi cura del programmatore assicurarsi che non vi siano inconsistenze, pena un probabile comportamento impredicibile del programma.

Sommario

La seguente tavola riassume lo scopo e la classe delle principali dichiarazioni

Dichiarazione	**Scopo**	**Classe**	**Visibilità**[a]
Fuori dai blocchi	Globale	`extern`	Ovunque
`extern` fuori dai blocchi	Globale	`extern`	Ovunque
`static` fuori dai blocchi	Globale	`static`	File
Inizio blocco	Locale	`auto`	Blocco
`register` inizio blocco	Locale	`register` o `auto`	Blocco
`static` inizio blocco	Locale	`static`	Blocco
`extern` inizio blocco	Locale	`extern`	Blocco
Nel prototipo	Locale	`auto`	Prototipo[b]
`register` nel prototipo	Locale	`register` o `auto`	Prototipo[b]

[a] Se non nascosta.
[b] Corpo della funzione nel caso della definzione di funzione.

Se una variabile viene inizializzata nella dichiarazione il valore fornito le viene assegnato una

o più volte a seconda della sua classe di memorizzazione:

Classe		Inizializzazione
statica	⇒	Una sola volta quando viene creata
automatica	⇒	Ogni volta che viene creata

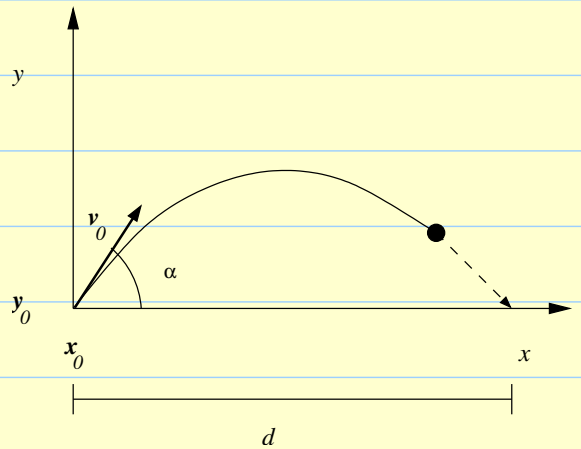
Se invece una variabile non viene inizializzata esplicitamentea nella dichiarazione questa viene inizializzata automaticamente o no a seconda della sua classe di memorizzazione:

Classe		Inizializzazione
statica	⇒	a zero quando viene creata
automatica	⇒	non inizializzata, valore indeterminato

2.23. Esempio: Distanza e tempo di caduta di un sasso in presenza di attrito (Rev. 2.1.1)

Come primo esempio di programmazione strutturata e utilizzo delle funzioni consideriamo un programma che determini la distanza di caduta ed il tempo di volo di un sasso lanciato in aria in presenza di attrito viscoso. Il realtà come vedremo, proprio grazie all'utilizzo di funzioni, il programma avrà una validità più generale.

Come al solito per prima cosa bisogna individuare un algoritmo che risolva il problema. Fissato un sistema di riferimento come in figura



il moto del sasso è descritto dalle equazioni

$$x(t) - x_0 = \frac{v_x}{\gamma}\left(1 - e^{-\gamma t}\right)$$

$$y(t) - y_0 = \frac{v_y}{\gamma}\left(1 - e^{-\gamma t}\right) - \frac{1}{2} g\, t^2$$

dove γ è il coefficiente di attrito, g è l'accelerazione di gravità, (x_0, y_0) sono le coordinate del punto da cui viene lanciato il sasso prese come origine del sistema di riferimento, e (v_x, v_y) sono le componenti della velocità iniziale del sasso lungo i due assi:

$$v_x = v_0 \cos\alpha, \qquad v_y = v_0 \sin\alpha$$

dove v_0 il modulo della velocità e α l'angolo che la velocità forma con l'asse orizzontale x.

La distanza d sull'asse x dal punto iniziale (x_0, y_0) a cui il sasso cade a terra è

$$d = x(t^*) - x_0$$

dove t^* è il tempo di "volo" dato dalla soluzione non nulla dell'equazione

$$y(t^*) - y_0 = 0, \qquad t^* \neq 0$$

Chiaramente questa equazione ammette sempre la soluzione banale $t = 0$ che corrisponde all'istante iniziale.

Da questa analisi segue che la soluzione del problema può essere divisa in due problemi più semplici. Il primo, ed anche il più semplice, è quello di scivere le funzioni che forniscono la traiettoria del sasso. Il secondo, più complesso, è invece quello di determinare lo zero o radice di una funzione di una variabile data. I due problemi sono sostanzialmente indipendenti per cui possono essere risolti separatamente e poi "rimessi insieme" per fornire la soluzione del problema originale. Questa analisi e decomposizione di un problema in problemi più semplici che possono essere risolti separatamente è alla base della programmazione strutturata.

La scrittura delle funzioni che forniscono la traiettoria del sasso non pone gravi problemi, per cui ci resta solo da trovare un algoritmo per determinare numericamente la radice di una funzione di una variabile. Per prima cosa è chiaro che a meno di eventi molto fortunati non è possibile determinare numericamente la radice esatta di una funzione. Il motivo è semplice tutte le operazioni sul computer sono effettuate con un numero finito di bits e quindi con una precisione finita. Questo vuol dire che due numeri che differiscono per un valore inferiore alla precisione non sono distinguibili numericamente. Questo sembrerebbe porre la parola fine al nostro problema, ma in realtà non è così. Infatti è vero che non si può determinare numericamente il valore esatto della radice, ma è sempre possibile determinare numericamente un intervallo che contenga la radice. La lunghezza dell'intervallo rappresenterà la precisione numerica con cui è stata determinata la radice: quanto più piccolo è l'intervallo tanto migliore è la stima numerica ottenuta. Chiaramente l'intervallo non potrà mai essere inferiore alla precisione numerica del computer.

Vi sono vari algoritmi per determinare numericamente le radici di una funzione che differiscono per le strategie utilizzate e per le informazioni richieste sulla funzione, ma tutti sostanzialmente utilizzano valutazioni successive della funzione su una serie di valori della variabile indipendente scelti con qualche regola. L'algoritmo utilizzato dal programma seguente si basa sul metodo della bisezione. Questo assume che si conosca un intervallo $[t_1, t_2]$ della variabile indipendente che contenga la radice t^* cercata. Il valore numerico di t^* viene stimato dividendo ogni volta l'intervallo che contiene la radice in due intervalli uguali e scegliendo poi quello che contiene ancora la radice. L'operazione di divisione/scelta viene ripetuta fino a quando non si è raggiunta la lunghezza dell'intervallo, e quindi la precisione, desiderata.

La scrittura del programma può essere semplificata rappresentando in modo *schematico* la sua struttura logica. Lo schema sarà poi convertito in istruzioni in linguaggio C. Nel nostro caso lo schema logico è piuttosto semplice:

- Leggere i dati iniziali v_0 e α, controllando eventualmente la consistenza. Si deve lanciare in alto e non in basso o indietro.
- Determinare i due tempi positivi t_1 e t_2 non necessariamente vicini tali che $t_1 < t^* < t_2$.
- Stimare il valore della soluzione positiva dell'equazione

$$y(t^*) - y_0 = 0$$

  usando il metodo della bisezione partendo dall'intervallo $[t_1, t_2]$.
- Calcolare la distanza d utilizzando il valore numerico trovato per t^*.
- Output dei risultati.

Il seguente programma segue questo schema logico.

Il programma qui proposto assume la funzione $y(t)$ sia positiva per $t = 0^+$, dopo tutto il sasso deve essere lanciato verso l'alto, e chi vi sia una sola radice per $t > 0$. Inoltre dal momento che il valore di x_0 e y_0 è irrilevante ai fini del calcolo nel programma si assume $x_0 = y_0 = 0$.

Programma: `stone-frict.c`

```
/*
 * Descrizione : Distanza di caduta ed tempo di volo in
 *               di un sasso lanciato in aria.
 *               Attrito viscoso.
 *
 * Input       : modulo velocita' iniziale
 *               angolo di lancio in gradi
 *               coefficiente di attrito
 *
 * Output      : Scrive su stdout distanza di caduta ed
 *               il tempo di volo.
 *
 * $yaC-Primer: stone-frict.c v 1.0 11.02.05 AC $
 */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* Macros */
#define ACC_G    9.8                  /* acc. gravita' in m/s^2 */
#define T_2      100.                 /* primo valore t_2       */
#define GR_RD    0.0174532925199433   /* gradi -> rad           */

/* Prototipi delle funzioni */
```

```
double x(double time);
double y(double time);
double find_impact(double t_1, double t_2);

/* Variabili globali */
double  V_x, V_y;          /* componenti velocita  */
double     Gamma;          /* coefficiente attrito */

int main(void)
{
  double       v_0, alpha;       /* velocita iniziale   */
  double t_1, t_2, t_star;
  char           line[41];

  /* Modulo velocita', angolo di lancio e attrito */
  printf("\n");
  printf("Modulo velocita' (m/s): ");
  fgets(line, sizeof(line), stdin);
  sscanf(line, "%lf", &v_0);

  if (v_0 <= 0) {
    printf("\n Il modulo deve essere postivo!\n\n");
    return 1;
  }

  printf("Angolo (gradi)        : ");
  fgets(line, sizeof(line), stdin);
  sscanf(line, "%lf", &alpha);

  /* Componenti velocita' */
  alpha *= GR_RD;                       /* Gradi -> Radianti */
  V_x    = v_0 * cos(alpha);
  V_y    = v_0 * sin(alpha);

  if (V_y <= 0) {
    printf("\n Lancio verso il basso!\n\n");
    return 1;
  }

  printf("Coefficiente attrito  : ");
  fgets(line, sizeof(line), stdin);
  sscanf(line, "%lf", &Gamma);

  if (Gamma <= 0) {
    printf("\n Attrito negativo!\n\n");
    return 1;
  }

  /*
   * intervallo iniziale [t_1, t_2]: t_1 < t^* < t_2
   */
```

```
  /* t_2 :  y(t_2) < 0 */
  t_2 = T_2;
  while ( y(t_2) > 0.00 ) t_2 *= 2.0;

  /* t_1 :  y(t_1) > 0 */
  t_1 = 0.5 * t_2;
  while ( y(t_1) < 0.00 ) {
    t_2 = t_1;
    t_1 = 0.5 * t_2;
  }

  /* trova t^*: y(t^*) = 0 con t_1 < t^* < t_2 */
  t_star = find_impact(t_1, t_2);

  printf("\n");
  printf("Dist. impatto: %.4f\n", x(t_star));
  printf("Tempo di volo: %.4f\n\n", t_star);

  return 0;
}

/* Funzioni */

/* ----
 * x()
 */
double x(double t)
{
  double coef;
  coef = V_x / Gamma;

  return coef * (1.0 - exp(-Gamma * t));
}

/* ----
 * y()
 */
double y(double t)
{
  static double g = 0.5 * (double) ACC_G;
  double coef;
  double y;

  coef = V_y / Gamma;

  y = coef * (1.0 - exp(-Gamma *t)) - g * t * t;
  return y;
}

/* ----
 * find_impact()
 *
```

```
 * tempo di impatto con il metodo della bisezione
 *
 *  MAX_BIS : Numero massimo bisezioni
 *  XACC    : Precisione richiesta
 */
#define MAX_BIS 40
#define XACC (1.e-8)
double find_impact(double x_1, double x_2)
{
  int                i;
  double x_root, x_mid;
  double     y_mid, dx;

  dx     = x_2 - x_1;          /* intervallo iniziale   */
  x_root = x_2;                /* assunzione iniziale   */

  for (i = 0; i < MAX_BIS; ++i) {
    dx   *= 0.5;
    x_mid = x_root - dx;
    y_mid = y(x_mid);
    if (y_mid <= 0.0) x_root = x_mid;
    if (dx < XACC || y_mid == 0.00) return x_root;
  }

  printf("\n Error!!! Precisione troppo alta\n");
  exit (1);
}
#undef MAX_BIS
#undef XACC
```

Note sul programma: `stone-frict.c`

- ```
  #define ACC_G    9.8                  /* acc. gravita' in m/s^2 */
  #define T_2      100.                 /* primo valore t_2       */
  #define GR_RD    0.0174532925199433   /* gradi -> rad           */
  ```

  Il programma utilizza delle macros per il valore dell'accelerazione di gravità, il primo valore assegnato a t_2 e per il fattore di conversione da gradi a radianti. Le macros sono scritte utilizzando solo lettere maiuscole per distinguerle dalle variabili.

- ```
  double x(double time);
  double y(double time);
  double find_impact(double t_1, double t_2);
  ```

  Dichiarazione delle funzioni usate. La definizione delle funzioni è data più avanti nel file dopo la funzione `main()`. Siccome non viene specificata la classe di memorizzazione le funzioni sono in classe `extern` e quindi visibili anche in altri files sorgente che eventualmente componessero il programma. Se le si volesse render visibili solo in questo file sorgente si devono dichiarare in classe di memorizzazione `static`:

```
static double x(double time);
static double y(double time);
static double find_impact(double t_1, double t_2);
```

e definirle di conseguenza in classe `static`. In questo caso tuttavia il programma è composto da un solo file sorgente per cui non vi è differenza nell'uso di una o dell'altra classe.

- ```
  double  V_x, V_y;          /* componenti velocita  */
  double     Gamma;          /* coefficiente attrito */
  ```

Il programma utilizza variabili a scopo globale per i parametri delle funzioni $y(t)$ e $x(t)$. Per indicare che le variabili hanno uno scopo globale i loro identificatori iniziano con una lettera maiscola. Sebbene questo semplifichi il passaggio dei valori dei parametri alle funzioni, l'uso di variabili a scopo globale dovrebbe essere limitato allo stretto necessario perché la loro presenza rende i programmi meno trasparenti. In questo programma, sebbene non necessarie, le variabili globali vengono utilizzate al solo scopo di mostrarne l'uso.

- ```
  t_2 = T_2;
  while ( y(t_2) > 0.00 ) t_2 *= 2.0;
  ```

Subito dopo il lancio il valore di $y(t)$ è sicuramente positivo per cui all'estremo superiore dell'intervallo $[t_1, t_2]$ che contiene la radice il suo valore deve essere necessariamente negativo. Il valore di t_2 è trovato partendo dal valore iniziale `T_2` e raddoppiando ogni volta il valore fino a quando $y(t_2)$ diventa negativo.

- ```
  t_1 = 0.5 * t_2;
  while ( y(t_1) < 0.00 ) {
    t_2 = t_1;
    t_1 = 0.5 * t_2;
  }
  ```

Il valore dell'estremo inferiore dell'intervallo $[t_1, t_2]$ è preso uguale alla metà del valore dell'estremo superiore t_2. Se la scelta iniziale di t_2 è troppo grande può accadere che il valore di $y(t_1)$ sia negativo. In questo caso si dimezza il valore dell'estremo superiore t_2 fino a quando il valore di $y(t_1)$ diventa positivo, mantenendo però il valore di $y(t_2)$ negativo.

- ```
  t_star = find_impact(t_1, t_2);
  ```

La funzione `find_impact()` restituisce il valore del tempo di volo t^* stimato utilizzando il metodo della bisezione. La funzione prende come parametri gli estremi t_1 e t_2 dell'intervallo che contiene t^* ed il valore della componente verticale v_y della velocità con cui viene lanciato il sasso. Tutti i dettagli del metodo sono in questa funzione.

- ```
  #define MAX_BIS 40
  #define XACC (1.e-8)
  ```
  ```

Il numero massimo di bisezioni e la precisione richiesta sono definite con delle macro visibili solo nella funzione. Questo non è necessariamente il modo migliore perché potrebbe essere utile poterle specificare come parametri nella chiamata della funzione. Si è scelto questo modo per mostrare un esempio di macros limitate ad una zona di programma.

- `x_root = x_2;`

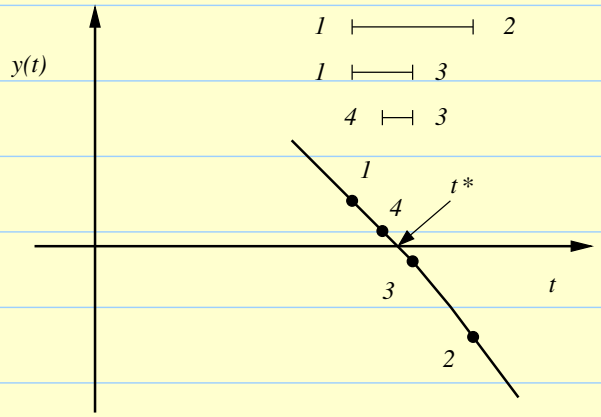
In questa formulazione dell'algoritmo di bisezione si usa come stima della radice l'estremo superiore dell'intervallo che contiene la radice esatta. Si prende quindi come stima iniziale della radice l'estremo superiore dell'intervallo iniziale $[x_1, x_2]$ fornito alla funzione.

- ```
  dx    *= 0.5;
  x_mid = x_root - dx;
  y_mid = y(x_mid);
  ```

Si calcola il punto di mezzo dell'intervallo `x_mid`, dimezzando contemporaneamente la sua lunghezza `dx`, ed il valore della funzione in questo punto `y_mid`.

- `if (y_mid <= 0.0) x_root = x_mid;`

La funzione $y(t)$ è sicuramente negativa all'estremo superiore dell'intervallo. Di conseguenza se anche nel punto di mezzo la funzione è negativa la soluzione cercata è nel semi-intervallo di lunghezza `dx` ed estremo superiore `x_mid`. Si sposta quindi l'estremo superiore dell'intervallo. La seguente figura mostra schematicamente le due prime iterazioni, indici 3 e 4.



Nella figura sono anche indicati dall'alto verso il basso le lunghezze degli intervalli successivi con i rispettivi estremi.

- `if (dx < XACC || y_mid == 0.00) return x_root;`

Se la lunghezza dell'intervallo è minore della precisione richiesta, o se si è trovata la soluzione esatta, si termina l'esecuzione della funzione restituendo il valore numerico stimato della radice.
  ```

- ```
  for (i = 0; i < MAX_BIS; ++i) {
  ...
  }

  printf("\n Error!!! Precisione troppo alta\n");
  exit (1);
  ```

  L'intervallo viene dimezzato per un numero massimo `MAX_BIS`. Se il numero di divisioni non è sufficiente per stimare il valore numerico della radice con la precisione richiesta viene stampato un messaggio sullo `stdout` e l'esecuzione dell'intero programma si interrompe.

- ```
  #undef MAX_BIS
  #undef XACC
  ```

  Le macros `MAX_BIS` e `XACC` sono limitate alla funzione `find_impact()` e quindi finita la funzione vengono cancellate. In questo caso non sarebbe necessario poiché il file sorgente termina dopo questa funzione, tuttavia suggeriamo di cancellare tutte le macros non più necessarie, anche nei casi in cui questo sia superfluo, in modo da ridurre al minimo le possibilità di conflitti tra macros qualora altre funzioni vengano aggiunte al file sorgente.

Osserviamo che l'uso delle funzioni ha reso il programma piuttosto flessibile, infatti se si volesse trattare il caso senza attrito o cambiare la forma dell'attrito sarebbe sufficiente modificare le funzioni $x(t)$ e $y(t)$ e non tutto il programma.

Compilazione e test del programma: Il programma usa le funzioni matematiche `sin()`, `cos()` ed `exp()` per cui va compilato con il *flag* "`-lm`" per aggiungere le librerie matematiche:

```
$ cc stone-frict.c -lm
```

Quando il programma viene eseguito si ha ad esempio

```
Modulo velocita' (m/s): 10
Angolo (gradi)        : 55
Coefficiente attrito  : 1.

Dist. impatto: 3.7074
Tempo di volo: 1.0395
```

Un modo piuttosto semplice di controllare la correttezza di un programma è quello di utilizzarlo per casi di cui si conosce il risultato. Ad esempio in assenza di attrito la distanza di caduta del sasso, per un valore fissato del modulo della velocità iniziale, deve essere la stessa per angoli simmetrici rispetto $45°$. Non è possibile tuttavia fornire al programma direttamente il valore $\gamma = 0$ perchè nelle funzioni $x(t)$ e $y(t)$ si otterrebbe il valore indeterminato $\frac{0}{0}$. Si può aggirare il problema verificando che la differenza delle distanze di caduta per due angoli simmetrici rispetto a $45°$, e per uno stesso valore del modulo della velocità iniziale, tende a zero al diminuire del valore γ utilizzato.

Esercizi

1. Modificare le funzioni $x(t)$ e $y(t)$ in modo che il programma fornisca il risultato esatto anche per $\gamma = 0$.
2. Considerare il caso di resistenza dell'aria costante.

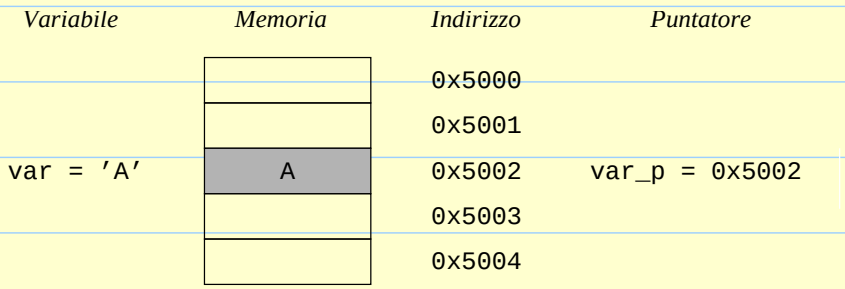
2.24. Puntatori (Rev. 2.1.2)

Quando un oggetto, come ad esempio una variabile, o una funzione vengono definiti il compilatore C gli assegna automaticamente uno spazio di memoria composto da un numero di unità di memoria sufficienti a contenerli. Una delle caratteristiche più potenti del linguaggio C è la possibilità di accedere ad ogni oggetto o funzione specificando direttamente l'indirizzo delle unità di memoria loro assegnate. Gli indirizzi di memoria sono assegnati automaticamente dal compilatore ed in genere non possono essere modificati, però possono essere manipolati mediante le variabili di tipo puntatore ad oggetto o funzione, o semplicemente *puntatori*. Il valore di un puntatore è l'indirizzo di memoria dell'oggetto o funzione a cui "punta" per questo motivo i puntatori sono anche chiamati variabili di indirizzo. La possibilità del linguaggio C di poter effettuare operazioni con i puntatori lo rende un linguaggio di programmazione molto potente e versatile.

Per illustrare il concetto di puntatore consideriamo la seguente dichiarazione

```
char var = 'A';
```

che crea una variabile di tipo `char` identificata dal nome `var` e gli assegna il valore '`A`'. Questo vuol dire che il compilatore associa all'identificatore `var` una zona di memoria di dimensioni sufficienti a contenere un oggetto di tipo `char`, in questo caso una sola unità di memoria, e vi memorizza il valore assegnato alla variabile:



In questo esempio il compilatore ha assegnato alla variabile `var` l'unità di memoria di indirizzo `0x5002` evidenziata in figura. I caratteri '`0x`' indicano che i valori sono espressi con digits esadecimali. Gli indirizzi di memoria sono usualmente espressi in notazione esadecimale perché ogni digit esadecimale rappresenta il valore di quattro digits binari successivi.

Consideriamo adesso il puntatore `var_p` di tipo `puntatore` a `char`. Per aumentare la chiarezza quando necessario per i puntatori saranno utilizzati identificatori contraddistinti dal suffisso "`_p`". Per il momento non ci preoccupiamo di come si definisca un puntatore ad un oggetto

di tipo `char` né di come gli si assegni il valore dell'indirizzo della variabile `var`, ma ci basta sapere che `var_p` è un puntatore a `char` e che quindi può contenere indirizzi di memoria di oggetti di tipo `char`. Se il valore del puntatore `var_p` è `0x5002`, l'indirizzo della locazione di memoria assegnata alla variabile `var`, allora `var_p` *punta* alla variabile `var` ed è possibile accedere al valore della variabile utilizzando sia il suo identificatore `var` che il puntatore alla variabile `var_p`.

Gli oggetti che possono essere definiti nel linguaggio C possono differire sia per tipo, ad esempio una variabile ed un array, che per dimensione, ad esempio tipo `float` e `double`. Nel caso delle funzioni queste possono differire anche nel numero e nel tipo di parametri formali, oltre che nel tipo del valore restituito. I puntatori invece indipendentemente dall'oggetto o funzione puntata contengono sempre e solo degli indirizzi di memoria per cui è necessario informare il compilatore di che tipo di oggetto o funzione si trova all'indirizzo di memoria indicato dal puntatore in modo che il contenuto della memoria possa essere interpretato correttamente. Diretta conseguenza di ciò è che un puntatore può puntare solo al tipo di oggetto o di funzione specificato nella sua dichiarazione. In altre parole un puntatore a tipo `int` può contenere solo l'indirizzo di memoria di variabili di tipo `int` e non, ad esempio, di una variabile di tipo `float`.

2.24.1. Operatori di referenza e dereferenza

Per accedere ad un oggetto o funzione tramite un puntatore o per conoscere l'indirizzo di memoria di un oggetto o funzione si utilizzano gli operatori unari di dereferenza e referenza

Operatore	Operazione
`*`	dereferenza
`&`	referenza

Entrambi gli operatori hanno associatività da destra e un livello di precedenza piuttosto alto. L'operatore unario di dereferenza "`*`" prende come operando un puntatore ed il suo valore è il contenuto della memoria puntata dal puntatore. L'operatore di referenza o indirizzo "`&`" prende come operando un oggetto o funzione ed il suo valore è l'indirizzo di memoria dell'oggetto o funzione. Di seguito sono riportati alcuni esempi nel caso in cui `var` sia una variabile e `ptr_p` un puntatore:

Espressione	Significato
`&var`	Puntatore alla variabile `var`
`*ptr_p`	Contenuto della memoria puntata da `ptr_p`
`&ptr_p`	Puntatore al puntatore `ptr_p`
`*var`	Illegale

I puntatori al pari di tutti gli altri oggetti hanno anche loro una zona di memoria assegnata in cui viene memorizzato il loro valore, di conseguenza l'espressione "`&ptr_p`" è perfettamente

lecita e fornisce l'indirizzo di memoria del puntatore `ptr_p`. L'espressione "`*var`" è invece illegale poiché `var` non è una variabile di tipo puntatore.

2.24.2. Operatori e puntatori

Gli altri operatori che possono essere usati con i puntatori sono gli operatori di assegnamento, addizione e sottrazione di interi, di relazione e di uguaglianza, `AND`, `OR` e `NOT` logici e conversione con tipi interi o puntatore a tipo diverso. Il significato di queste operazioni è piuttosto semplice. L'operatore di assegnamento "`=`" permette di assegnare il valore al puntatore. Aggiungere o sottrarre un intero `n` ad un puntatore `prt` a tipo `T` equivale invece a spostarsi in avanti o indietro nella memoria rispetto all'indirizzo di memoria "puntato" da `ptr` di un numero `n*sizeof(T)` di unità di memoria pari alla dimensione "`sizeof(T)`" del tipo `T` per il valore dell'intero `n`. Gli operatori di relazione, di uguaglianza e logici permettono di confrontare tra loro indirizzi di memoria, mentre il cast permette di convertire un puntatore a tipo `T` nel puntatore a tipo `T'` o ad un tipo intero e viceversa.

2.24.3. Dichiarazione di puntatori

La dichiarazione di un puntatore ad un oggetto o ad una funzione è sintatticamente uguale a quella dell'oggetto o della funzione con la sola differenza che l'identificatore viene preceduto dall'operatore di dereferenza "`*`". Ad esempio nel caso del puntatore `var_p` alla variabile

```
char var;
```

questo viene dichiarato come:

```
char *var_p;
```

Questa dichiarazione può essere letta come: "l'oggetto puntato dal puntatore `var_p` è di tipo `char`".

La precedente dichiarazione dichiara `var_p` come un puntatore a tipo `char` ma non gli assegna come valore l'indirizzo della variabile `var`, di fatto non gli assegna nessun valore,[7] per cui nonostante il suo nome il puntatore `var_p` non è un puntatore alla variabile `var` ma semplicemente un puntatore che può contenere l'indirizzo di memoria di una variabile di tipo `char`. Affinché `var_p` punti effettivamente alla variabile `var` bisogna assegnargli come valore l'indirizzo di memoria della varibile `var` mediante l'operatore "`&`":

```
var_p = &var;
```

Il puntatore `var_p` è adesso un puntatore alla variabile `var` per cui `*var_p` è il contenuto della zona di memoria assegnata alla variabile `var`, ossia il valore della variabile `var`. Il seguente programma illustra questo punto.

Programma: `pointer_var.c`

[7] Se il puntatore è dichiarato con scopo globale e non viene fornito un valore iniziale nella dichiarazione come tutte le variabili globali il puntatore viene inizializzato automaticamente a zero.

```
#include <stdio.h>

int main(void)
{
  int    var;     /* dichiara una variabile di tipo int      */
  int *var_p;     /* dichiara un puntatore ad un oggetto int */

  var_p = &var;   /* var_p punta a var */

  var = 7;        /* assegna un valore a var */
  printf("1./ Valori: var = %d -- *var_p = %d\n", var, *var_p);

  *var_p = 8;     /* cambia valore a var */
  printf("2./ Valori: var = %d -- *var_p = %d\n", var, *var_p);

  return 0;
}
```

Osserviamo che siccome il puntatore var_p è un puntatore alla variabile var l'istruzione

```
*var_p = 8;
```

è perfettamente equivalente all'istruzione

```
var = 8;
```

e quindi assegna il valore 8 alla variabile var. L'istruzione infatti può essere letta come: "assegna il valore 8 come contenuto della zona di memoria puntata da var_p". Questo è mostrato chiaramente dall'output del programma:

```
1./ Valori: var = 7 -- *var_p = 7
2./ Valori: var = 8 -- *var_p = 8
```

Più puntatori possono puntare allo stesso oggetto. Nel seguente programma entrambi i puntatori uno_p e due_p puntano alla stessa variabile var:

Programma: `pointer2_var.c`

```
#include <stdio.h>

int main(void)
{
  int    var;     /* variabile di tipo int             */
  int *uno_p;     /* puntatore ad un oggetto int       */
  int *due_p;     /* altro puntatore ad un oggetto int */

  uno_p = &var;   /* uno_p punta a var       */
  due_p = uno_p;  /* anche due_p punta a var */

  var = 7;
  printf("1./ Valori: var = %d -- *uno_p = %d -- *due_p = %d\n",
         var, *uno_p, *due_p);
```

```
    *uno_p = 8;     /* cambia valore a var */
    printf("2./ Valori: var = %d -- *uno_p = %d -- *due_p = %d\n",
           var, *uno_p, *due_p);

    *due_p = 9;     /* cambia valore a var */
    printf("3./ Valori: var = %d -- *uno_p = %d -- *due_p = %d\n",
           var, *uno_p, *due_p);

    return 0;
}
```

Quando il programma viene eseguito si ha il seguente output:

```
1./ Valori: var = 7 -- *uno_p = 7 -- *due_p = 7
2./ Valori: var = 8 -- *uno_p = 8 -- *due_p = 8
3./ Valori: var = 9 -- *uno_p = 9 -- *due_p = 9
```

da cui si vede chiaramente che sia `*uno_p` che `*due_p` fornisono il valore della variabile `var`.

I puntatori sono classificati come tipi interi, dopotutto contengono indirizzi di memoria, tuttavia le loro dimensioni dipendono da come viene gestita ed indirizzata la memoria con il risultato che le dimensioni possono variare da un sistema all'altro. Inoltre su uno stesso sistema le dimensioni dei puntatori possono differire a seconda del tipo di oggetto a cui puntano. Ad esempio i puntatori ad oggetti di tipo dati, come variabili o arrays, possono essere più corti o più lunghi dei puntatori a tipo funzione. Il linguaggio C non richiede che vi sia necessariamente una relazione tra le dimensioni dei puntatori e quella dei tipi interi anche se usualmente si assume che le dimensioni del tipo `long` siano almeno pari a quelle del tipo puntatore, cosa che usualmente è. Lo Standard C99 introduce i tipi `intptr_t` e `uintptr_t` definiti nel file di header `inttypes.h` di dimensioni sufficienti a contenere sia un tipo intero che un tipo puntatore.

2.24.4. Puntatori a tipo array

La dichiarazione di un puntatore ad un oggetto può richiedere l'uso delle parentesi "`()`" per la corretta interpretazione della dichiarazione anche se queste non sono necessarie nella dichiarazione dell'oggetto. Questo è il caso ad esempio dei puntatori ad arrays. Consideriamo infatti la seguente dichiarazione di un array di 5 elementi di tipo `int`

```
int a[5];
```

La regola dice che per dichiarare un puntatore ad array di `5` elementi di tipo `int` bisogna aggiungere l'operatore di dereferenza "`*`" prima dell'identificatore:

```
int *p[5];
```

Tuttavia l'operatore di indice "`[]`" ha un livello di precedenza più alto dell'operatore di dereferenza "`*`" di conseguenza la dichiarazione viene interpretata come la dichiarazione di un array di `5` elementi di tipo `int *`. In altre parole l'identificatore `p` viene associato ad un

array i cui elementi sono puntatori a tipo `int`, si ha così un *array di puntatori* al posto di un puntatore ad un array.

Per dichiarare un puntatore ad un array di `5` elementi di tipo `int` bisogna alterare la precedenza degli operatori utilizzando le parentesi "`()`" per "forzare" l'applicazione dell'operatore di dereferenza prima di quello di indice. La dichiarazione di un puntatore ad un array di `5` elementi di tipo `int` è quindi

```
int (*p)[5];
```

Le parentesi forniscono ora la corretta interpretazione della dichiarazione:

`(*p)`	⇒	l'oggetto puntato dal puntatore `p`
`(*p)[5]`	⇒	è un array di `5` elementi
`int (*p)[5]`	⇒	di tipo `int`

Il seguente programma illustra l'uso del puntatore ad array

Programma: `pointer_array.c`

```
#include <stdio.h>

int main(void)
{
  int       i;
  int    a[5];          /* array               */
  int (*p)[5];          /* puntatore ad array */

  p = &a;               /* p punta ad a        */

  for (i = 0; i < 5; ++i)
    (*p)[i] = 2 * i+1;

  printf("i    a[i] (*p)[i]\n");
  printf("-----------------\n");
  for (i = 0; i < 5; ++i)
    printf("%d     %d      %d \n", i, a[i], (*p)[i]);

  return 0;
}
```

L'output di questo programma è

```
i    a[i] (*p)[i]
-----------------
0     1      1
1     3      3
2     5      5
3     7      7
4     9      9
```

Osserviamo che le parentesi devono usate anche quando si accede all'array tramite il puntatore:

```
(*p)[i] = 2 * i + 1;

printf("%d      %d      %d \n", i, a[i], (*p)[i]);
```

altrimenti a causa nuovamente della precedenza dell'operatore "`[]`" sull'operatore "`*`" la scrittura `*p[i]` verrebbe interpretata come il contenuto dell'indirizzo di memoria puntato dal puntatore `p[i]`, e non come l'elemento `i` dell'array puntato dal puntatore `p`.

La dimensione dell'array può essere omessa nella dichiarazione del puntatore ad array. Ad esempio la dichiarazione

```
int (*p)[ ];
```

dichiara il puntatore `p` ad un array di tipo `int` *incompleto*. Sebbene non sia possibile dichiarare oggetti di tipo incompleto, ad esempio un array senza specificare la sua dimensione, perché la loro dimensione non è conosciuta, puntatori ad oggetti di tipo incompleto possono essere creati ed usati. Il linguaggio C permette infatti dichiarazioni incomplete se le informazioni mancanti vengono fornite successivamente. In questo caso l'informazione sulla dimensione viene data assegnando l'indirizzo di memoria di un array di tipo `int`, e quindi di dimensione nota, al puntatore `p`. Un puntatore ad un tipo incompleto è quindi un puntatore al tipo per cui il puntatore `p` dell'esempio è un puntatore al tipo array di tipo `int` e come tale può essere usato con arrays di tipo `int` di dimensione qualsiasi.

Al contrario se nella dichiarazione del puntatore viene specificata la dimensione dell'array il puntatore è un puntatore ad un tipo completo e quindi può essere usato solo per puntare a quel tipo completo, ossia arrays di tipo `int` di dimensione uguale a quella specificata nella dichiarazione. Ad esempio se nel programma precedente si fosse usata la dichiarazione

```
int (*p)[6];
```

il compilatore avrebbe segnalato una inconsistenza.

2.24.5. Puntatori a funzione

La dichiarazione di un puntatore a funzione si effettua con modalità analoghe a quelle dei puntatori ad oggetti, ossia precedendo nel prototipo l'identificatore della funzione con l'operatore di dereferenza "`*`". Ad esempio l'istruzione

```
double (*fdd_p)(double x);
```

dichiara la variabile `fdd_p` di tipo puntatore a "funzione di tipo `double` con un parametro di tipo `double`". Come per la dichiarazione di una funzione gli identificatori dei parametri formali non sono necessari e spesso sono omessi nelle dichiarazioni di puntatori a funzione. La dichiarazione precedente viene quindi scritta come:

```
double (*fdd_p)(double);
```

Nella dichiarazione del puntatore a funzione le parentesi "`()`" intorno all'identificatore del puntatore sono necessarie poiché l'operatore di funzione "`f(...)`" ha un livello di precedenza più alto dell'operatore di dereferenza "`*`" per cui l'istruzione

```
double *fdd_p(double);
```

viene interpretata come la dichiarazione della funzione `fdd_p()` di tipo `double *` che prende un parametro di tipo `double` e restituisce come valore un puntatore a `double`.

Una variabile di tipo puntatore a funzione può puntare solo alle funzioni il cui prototipo coincide con quello nella dichiarazione del puntatore, questo vuol dire che `fdd_p` può puntare ad una qualsiasi funzione il cui prototipo sia

```
double function_name(double);
```

Ad esempio può puntare alla funzione `sin()`:

```
double sin(double x);
```

ma non alla funzione `pow()` poiché il suo prototipo è

```
double pow(double x, double y);
```

Nella dichiarazione di un puntatore a "funzione di tipo `T`" i parametri formali della funzione possono essere omessi del tutto nel qual caso si ottiene la dichiarazione di un puntatore ad una funzione di tipo `T` incompleta. Analogamente ai puntatori ad array incompleti un puntatore a funzione di tipo `T` incompleta può essere utilizzato con tutte le funzioni che restituiscono un valore di tipo `T` indipendentemente dal numero e tipo dei loro parametri formali. Le informazioni mancanti, numero e tipo dei parametri, vengono infatti fornite con l'assegnazione dell'indirizzo di memoria di una funzione di tipo `T` al puntatore. Ad esempio l'istruzione

```
double (*fd_p)();
```

dichiara la variabile `fd_p` di tipo "puntatore a funzione di tipo `double`" che può essere utilizzata per puntare, ad esempio, sia alla funzione `sin()` che alla funzione `pow()` ma non alla funzione `printf()` poiché questa restituisce un valore di tipo `int`

```
int printf(const char *format, ...);
```

Per assegnare il valore a una variabile di tipo puntatore a funzione si può usare l'operatore di indirizzo "`&`" applicato all'identificatore della funzione. Tuttavia nel caso delle funzioni questo può essere omesso poiché quando l'identificatore di una funzione di tipo `T`, o in generale una qualsiasi espressione del tipo "funzione che ritorna il tipo `T`", non è usato per effettuare una chiamata di funzione, ossia non è seguito dalla parentesi tonda "`(`", o non è argomento dell'operatore di indirizzo "`&`" o dell'operatore `sizeof` viene convertito automaticamente in puntatore a funzione di tipo `T`. Di conseguenza le istruzioni

```
double (*fdd_p)(double);

fdd_p = &sin;     /* Assegnazione esplicita di puntatore */
fdd_p = sin;      /* Conversione implicita a puntatore   */
```

sono perfettamente equivalenti ed assegnano entrambi l'indirizzo della funzione `sin()` al puntatore `fdd_p`. L'assegnazione

```
fdd_p = pow;
```

è invece illegale perché la funzione `pow()` di tipo `double` ha due parametri, e non uno come richiede la dichiarazione del puntatore.

L'istruzione

```
fdd_p = &sin(2.5);
```

è invece illegale perché la chiamata di funzione ha la precedenza sull'operatore di indirizzo e quindi l'istruzione assegnerebbe l'indirizzo del valore di `sin(2.5)` alla variabile di tipo puntatore `fdd_p`, il che ovviamente non ha nessun senso.

L'operatore di dereferenza "`*`" applicato ad un puntatore a funzione restituisce la funzione, per cui se `fd_p` punta alla funzione `sin()` l'espressione

```
(*fd_p)(2.5)
```

ha come valore il valore di `sin(2.5)`. Le parentesi sono necessarie altrimenti, poiché la chiamata di funzione ha la precedenza sull'operatore "`*`", l'espressione

```
*fd_p(2.5)
```

verrebbe interpretata come il contenuto della locazione di memoria di indirizzo `fd_p(2.5)`, il che potrebbe avere conseguenze disastrose sul programma. Riassumendo quindi se `fd_p` è un puntatore a funzione dichiarato ad esempio come

```
double (*fd_p)();
```

allora

`fd_p`	⇒	puntatore alla funzione
`*fd_p`	⇒	la funzione
`(*fd_p)(x)`	⇒	chiamata alla funzione

Nello Standard C un'espressione del tipo "puntatore a funzione" può essere utilizzata in una chiamata a funzione senza l'operazione esplicita di derefererenza. Di conseguenza l'operatore di dereferenza "`*`" può essere omesso quando si effettua una chiamata a funzione tramite il puntatore a funzione, per cui l'istruzione

```
fd_p(2.5)
```

è perfettamente equivalente a

```
(*fd_p)(2.5)
```

il che semplifica molto la scrittura dei programmi.

Il seguente semplice programma illustra l'uso dei puntatori a funzioni.

Programma: `pointer_func.c`

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

double fun(double);

int main(void)
{
  double (*fd_p)();

  fd_p = fun;
  printf("x = %.3f -> x^2 =  %.3f\n", 2., (*fd_p)(2.));
  printf("x = %.3f -> x^2 =  %.3f\n", 3., fd_p(3.));

  fd_p = pow;
  printf("x = %.3f -> x^2 =  %.3f\n", 4., (*fd_p)(4.,2.));
  printf("x = %.3f -> x^2 =  %.3f\n", 5., fd_p(5.,2.));
  return 0;
}

double fun(double x)
{
  return(x*x);
}
```

Quando il programma viene eseguito si ottiene

```
x = 2.000 -> x^2 =  4.000
x = 3.000 -> x^2 =  9.000
x = 4.000 -> x^2 =  16.000
x = 5.000 -> x^2 =  25.000
```

da cui risulta evidente che la scrittura `(*fp)(3.)` e `fp(3.)` sono perfettamente equivalenti.

2.24.6. Qualificatore `const`

Nella dichiarazione di puntatori a variabili è possibile utilizzare il qualificatore `const`, tuttavia siccome nella dichiarazione di un puntatore si ha a che fare sia con il valore della variabile che con il suo indirizzo bisogna specificare se è costante il valore della variabile, del puntatore o di entrambi. Per chiarire questo punto consideriamo la seguenti dichiarazione

```
const int const_var = 7;
```

che dichiara la variabile `const_var` di tipo `int` il cui valore non può essere cambiato.

Se l'identificatore della variabile viene preceduto dall'operatore di dereferenza "`*`", secondo i dettami della regola per dichiarare un puntatore, si ottiene la dichiarazione

```
const int *ptr_to_const;
```

che letteralmente significa: "l'oggetto puntato da `ptr_to_const` è di tipo `const int`". Questa dichiarazione dichiara quindi un puntatore ad una variabile di tipo `const int`. Di conseguenza l'istruzione

```
ptr_to_const = &const_var;
```

è legale perché il puntatore non è dichiarato costante e quindi il suo valore può essere cambiato, mentre l'istruzione

```
*ptr_to_const = 8;
```

è illegale perché cambia il valore della variabile di tipo `const int` puntata da `ptr_to_const`.

Per dichiarare un puntatore costante, ossia un puntatore il cui valore non può essere cambiato, bisogna utilizzare la dichiarazione seguente

```
int *const const_ptr = &var;
```

che dichiara che il puntatore costante `const_ptr` punta all'oggetto di tipo `int` che trova in `var`. Il valore del puntatore non può essere cambiato per cui il puntatore va inizializzato nella sua dichiarazione, in questo caso con l'indirizzo della variabile `var`. Chiaramente la variabile `var` deve esistere per cui l'ordine delle dichiarazioni conta. Ad esempio le seguenti dichiarazioni

```
int var;
int *const const_ptr = &var;
```

e

```
int *const const_ptr = &var;
int var;
```

non sono equivalenti, ed in particolare il secondo ordinamento produce un errore in compilazione.

Dal momento che il valore del puntatore `const_ptr` è costante l'istruzione

```
const_ptr = &i;
```

con `i` una variabile di tipo `int` è illegale, tuttavia l'oggetto puntato da `const_var` non è dichiarato costante per cui il suo valore può essere cambiato e quindi l'istruzione

```
*const_ptr = 8;
```

è perfettamente legale.

Infine se si vuole che sia il valore del puntatore che quello dell'oggetto puntato non possano essere cambiati bisogna dichiarare costanti sia il puntatore che l'oggetto, come nella dichiarazione seguente:

```
const int *const const_ptr_to_const = &var;
```

che dichiara il puntatore costante `const_ptr_to_const` che punta all'oggetto di tipo `const int` contenuto nella variabile `var` di tipo `const int`.

2.24.7. Puntatore generico `void *`

A volte, soprattutto nella scrittura delle funzioni, si ha la necessità di poter disporre di un puntatore ad oggetti di tipo generico che possa essere eventualmente convertito in un puntatore ad un tipo qualunque `T`. Questo è ad esempio il caso di funzioni che possono accettare come parametri formali puntatori ad oggetti di qualsiasi tipo. Il Traditional C usava per questo scopo il tipo `char *`, tuttavia questo creava il problema che non vi era una chiara distinzione tra puntatori generici e puntatori ad oggetti. Per ovviare a questa limitazione lo Standard C ha introdotto il tipo `void *` che ha la stessa rappresentazione del tipo `char *` per compatibilità con il Traditional C ma è trattato diversamente da un puntatore normale per permettere al compilatore di distinguere tra puntatori ad oggetti e puntatori generici. Ad esempio un puntatore di tipo `void *` non può essere dereferenziato con l'operatore "`*`" né può comparire come operando degli operatori di addizione "`+`" e sottrazione "`-`".

La dimensione di un puntatore di tipo `void *` o di tipo `char *` può essere maggiore di quella degli altri puntatori, ed inoltre possono essere rappresentati in modo differente dagli altri puntatori.

Il tipo `void *` viene considerato né come puntatore ad oggetto né come puntatore a funzione. Tuttavia ogni puntatore ad oggetto, anche se di tipo incompleto, può essere convertito a puntatore di tipo `void *` e viceversa senza perdita di informazioni. In altre parole lo Standard C assicura che un puntatore ad un oggetto mantiene il suo valore se convertito a tipo `void *` e poi riconvertito nel tipo originale. Questo non accade invece con i puntatori a funzione perché non è garantito che la dimensione dei puntatori a funzione non sia maggiore della dimensione di un puntatore di tipo `void *`, anche usualmente non lo è, per cui la trasformazione a `void *` e viceversa può causare una perdita di informazioni.

La possibilità di poter definire puntatori di tipo generico aumenta la flessibilità delle funzioni perché permette di lasciare indefinito il tipo dei parametri formali. Quando una funzione ha un parametro formale che può essere un puntatore ad oggetti di tipo `T` diversi il parametro deve essere dichiarato di tipo `void *` nel prototipo della funzione. Se infatti venisse dichiarato di un tipo qualsiasi diverso da `void *` l'argomento corrispondente nella chiamata a funzione deve necessariamente essere del tipo dichiarato poiché lo Standard C non permette in genere di assegnare il valore di un puntatore a tipo `T` ad un puntatore di tipo `T'` diverso da `T`. Un esempio è il parametro corrispondente alla direttiva di conversione "`%p`" nelle funzioni di output `printf()`, `fprintf()` e `sprintf()` che deve essere di tipo `void *` per permettere alle funzioni scrivere il valore del puntatore indipendentemente dal tipo oggetto o funzione puntato.

Dal momento che nello Standard C il tipo `void *` può essere utilizzato come puntatore ad un oggetto generico è possibile definire funzioni di tipo `void *`, come vedremo ad esempio trattando le funzioni di gestione della memoria, il cui valore è un indirizzo di memoria che può contenere un oggetto di tipo `T` qualsiasi. Non esiste invece un puntatore a funzione generico.

2.24.8. Puntatore nullo `NULL`

Un'altra necessità che capita spesso è quella di avere un puntatore che non punta a nulla. Per questo scopo il C fornisce il puntatore nullo indicato dalla macro `NULL` che punta a nessun

oggetto o funzione. La macro `NULL` è definita nei files di header `stddef.h` o `stdio.h` come la costante "`0`", oppure come la costante "`0L`" se i puntatori sono più grandi del tipo `int`, o infine come "`(void *) 0`" a seconda dei sistemi.

Il puntatore nullo ha il valore *falso* in ogni espressione booleana per cui ad esempio il test

```
if (ptr != NULL) ....
```

può essere scritto anche

```
if (ptr) ....
```

L'uso di puntatori con valori illegali, ossia valori non nulli ma che non puntano a nessun oggetto o funzione valida, può causare una errore e l'arresto dell'esecuzione del programma. Questi errori possono essere molto difficili da individuare e correggere, di conseguenza è un buon stile di programmazione quello di assicurarsi che il valore dei puntatori sia `NULL` quando non sono utilizzati per puntare ad oggetti o funzioni.

2.24.9. Operatore di assegnamento e conversione

L'operatore di assegnamento "`=`" necessita di qualche parola in più. Lo Standard C richiede che nell'assegnazione

```
ptr = expression
```

con `ptr` puntatore a tipo `T` l'espressione `expression` sia

tipo puntatore `ptr`	**tipo espressione** `expression`
puntatore a oggetto tipo `T`	costante 0 puntatore ad oggetto di tipo `T` puntatore ad oggetto di tipo `T'` diverso da `T` ma compatibile. `void *`
puntatore a funzione tipo `T`	costante 0 puntatore a funzione di tipo `T` puntatore a funzione di tipo `T'` diverso da `T` ma compatibile.
`void *`	costante 0 puntatore ad oggetto di tipo `T`, eventualmente incompleto `void *`

Nel caso in cui `ptr` sia un puntatore ad oggetto di tipo `T` e `expression` sia di tipo `void *` il tipo `T` può essere un oggetto di tipo incompleto, ad esempio di tipo array `(*)[]`.

In tutti gli altri casi lo Standard C non accetta l'assegnazione. Di conseguenza se è necessario effettuare un assegnazione tra due puntatori a tipi non compatibili tra loro, ad esempio

assegnare il valore di un puntatore a tipo `int` ad un puntatore a tipo `char`, bisogna effettuare un cast esplicito.

```
char *char_p;
int  *int_p;

char_p = int_p            /* Illegale nello  Standard C */
char_p = (char *) int_p   /* Permesso nello  Standard C */
```

Non entreremo nel dettaglio di quali sono i tipi compatibili tra loro ma suggeriamo di utilizzare un cast esplicito ogni qual volta non si è sicuri del risultato.

Il Traditional C, ed alcuni compilatori non Standard ISO C, sono meno restrittivi per cui non è necessario effettuare cast espliciti nelle assegnazioni.

Nota: Per assegnare un valore di tipo "puntatore a `volatile T`" ad un variabile di tipo "puntatore a tipo `T`" è necessario effettuare un cast esplicito a `(T *)`, come mostra il seguente stralcio di programma

```
volatile int *p_v;
int  *p ;

p_v = p;                  /* Legale   */
p   = p_v;                /* Illegale */
p   = (int *) p_v;        /* Legale   */
```

I puntatori a tipo `volatile T` sono usati per accedere a zone di memoria che contengono informazioni modificate dall'hardware del computer, ad esempio lo stato o il tipo dei vari devices del computer.

2.24.10. Alcune considerazioni sui puntatori

I puntatori sono variabili che contengono indirizzi di memoria per cui sembrerebbe naturale che tutti i puntatori siano della stessa dimensione ed utilizzino la stessa rappresentazione. Spesso in effetti si assume che sia effettivamente così per cui le limitazioni nelle assegnazioni e conversioni appaiono molto strane, se non incomprensibili.

In realtà questa assunzione è erronea perché le modalità di accesso ed indirizzamento delle unità di memoria dipendono dall'architettura del computer. Il risultato è che non solo la dimensione e la rappresentazione delle variabili utilizzate per memorizzare un indirizzo di memoria possono variare da sistema a sistema ma anche che su uno stesso sistema possano dipendere dal tipo di informazioni contenute nella memoria.

Per potersi adattare facilmente alle diverse rappresentazioni utilizzate dai vari sistemi il linguaggio C non richiede che tutti i puntatori siano della stessa dimensione ed utilizzino la stessa rappresentazione lasciando quindi un certo grado di arbitrarietà nelle loro proprietà. Conseguenza di ciò è che

- Non vi è una relazione tra la dimensione dei tipi interi e quella dei puntatori, per cui la dimensione di un puntatore non è necessariamente quella usata per il tipo `int`. Su

molti sistemi la dimensione di un puntatore è la stessa di un `long`, ma anche questo non è necessariamente vero.

- Puntatori di tipo `void *` e `char *` possono avere dimensioni più grandi degli altri puntatori ed essere rappresentati in modo differente da questi ultimi.
- Puntatori ad oggetti e puntatori a funzione possono avere rappresentazioni e dimensioni differenti
- Usualmente i puntatori a funzioni non sono di dimensione maggiore dei puntatori di tipo `void *`, ma anche questo non è necessariamente vero.

Da qui l'origine delle limitazioni nelle assegnazioni e conversioni con puntatori.

A causa di queste arbitrarietà nelle proprietà dei puntatori si suggerisce di utilizzare sempre un cast esplicito quando si effettuano conversioni tra puntatori di tipo diverso, o con tipi interi. Attenzione particolare inoltre deve essere posta negli argomenti delle funzioni affinché questi siano sempre del tipo corretto che la funzione si aspetta, utilizzando se necessario un cast esplicito. In caso contrario siccome l'utilizzo dei puntatori da accesso alla memoria ad un livello piuttosto basso non solo i risultati potrebbero essere disastrosi ma l'individuazione degli eventuali errori potrebbe essere piuttosto difficile.

2.25. Puntatori come parametri di funzione (Rev. 2.1.1)

Nel linguaggio C il parametri delle funzioni sono passati per valore per cui in una chiamata a funzione il valore degli argomenti, siano essi costanti, variabili od espressioni, viene valutato ed assegnato ai corrispondenti parametri della funzione. La conseguenza più evidente del passaggio per valore è che non è possibile cambiare il valore di una variabile data come argomento della chiamata a funzione dall'interno della funzione. La funzione conosce infatti solo il valore della variabile tramite il suo parametro a cui il valore è stato assegnato ma non la variabile, per cui non può cambiarne il valore.

Se fosse necessario che la funzione cambi il valore di una sola variabile si potrebbe aggirare il problema sfruttando il valore restituito dalla funzione

```
var = func(var,...);
```

In questo modo infatti il valore della variabile `var` verrebbe utilizzato dalla funzione `func()` per i suoi scopi e, finita l'esecuzione della funzione, cambiato nel valore restituito dalla funzione. Tuttavia se è necessario cambiare il valore di più di una variabile questa strategia è chiaramente inapplicabile poiché una funzione può restituire al massimo un solo valore. Il passaggio per valore può sembrare quindi a prima vista molto limitativo. In realtà non è così poiché il problema può essere risolto facilmente utilizzando i puntatori per passare alle funzioni non il valore delle variabili a cui si deve cambiare il valore, ma bensì il loro indirizzo di memoria. In questo modo la funzione "conosce" dove è scritto il valore della variabile e può quindi facilmente cambiarlo.

Per illustrare questa strategia supponiamo ad esempio di volere una funzione che ogni volta che viene chiamata aggiunga alla variabile `sum` il valore della variabile `var` data come argomento

della funzione. In questo caso si deve modificare il valore di una sola variabile, la variabile `sum`, per cui si potrebbe utilizzare la funzione `sum_var()` seguente

Programma: `sum_var1.c`

```
#include <stdio.h>

int sum_var(int x, int y);

int main(void)
{
  int var;
  int sum;

  sum = 0;
  for (var = 1; var < 7; ++var) {
    sum = sum_var(var, sum);
    printf("var: %d -> somma: %d\n", var, sum);
  }

  return 0;
}

int sum_var(int x, int y)
{
  return (x + y);
}
```

Supponiamo adesso che il valore della variabile `var` debba essere aggiunto a quello della variabile `sum` solo nel caso in cui sia positivo. La funzione `sum_var()` può essere modificata facilmente per assolvere questo compito:

```
int sum_var(int x, int y)
{
  if (x > 0) return (x + y);
  return y;
}
```

In questo caso tuttavia potrebbe essere utile avere anche un contatore `count` del numero di valori sommati, ad esempio per farne la media. Chiaramente non è possibile passare `count` come argomento della funzione poiché il suo valore non può essere modificato dalla funzione né tantomeno utilizzare il valore della funzione essendo questo è già usato per la somma. È possibile tuttavia passare alla funzione l'indirizzo di memoria della variabile `count` in modo che la funzione conoscendo dove è scritto il valore della variabile possa eventualmente modificarlo, come mostrato nel programma seguente.

Programma: `sum_var2.c`

```
#include <stdio.h>

int sum_var(int x, int y, int *count_p);
```

```
int main(void)
{
  int sum, count;
  int i;
  int a[ ] = { 1, -2, -3, 4, 5, 6, 7};

  sum   = 0;
  count = 0;
  for (i = 0; i < 7; ++i) {
    sum = sum_var(a[i], sum, &count);
    printf("var: %3d -> somma: %3d -> count: %3d\n", a[i], sum, count);
  }

  return 0;
}

int sum_var(int x, int y, int *count_p)
{
  if (x > 0) {
    ++*count_p;
    return (x + y);
  }
  return y;
}
```

In questo esempio il valore da sommare al valore della variabile `sum` è il valore dell'elemento di indice `i` dell'array `a`.

Il il terzo argomento della funzione `sum_var()` è un `puntatore` a tipo `int` per cui il prototipo della funzione è

```
int sum_var(int x, int y, int *count_p);
```

Siccome nella dichiarazione di funzione i nomi dei parametri formali nel prototipo della funzione non sono obbligatori si sarebbe potuto utilizzare anche la forma compatta, ma meno chiara,

```
int sum_var(int, int, int *);
```

o un una qualsiasi forma intermedia tra le due.

Nella chiamata della funzione

```
sum = sum_var(a[i], sum, &count);
```

si utilizza l'operatore di referenza "`&`" per passare alla funzione l'indirizzo di memoria della variabile `count`, che viene assegnato al parametro `count_p` della funzione, e non il suo valore.

Se il valore di `a[i]` viene aggiunto somma il valore della variabile `count` viene aumentato di uno all'interno della funzione `sum_var()` con l'istruzione

```
++*count_p;
```

La funzione `sum_var()` conosce infatti il puntatore `count_p` alla variabile `count` di conseguenza per accedere al valore della variabile `count` e modificarlo si deve utilizzare l'operatore di dereferenza "`*`". La precedente istruzione prende quindi il valore della variabile puntata da `count_p` (valore di `count`), lo aumenta di `1` e riassegna il valore così ottenuto alle unità di memoria indicate da `count_p`, in forma più esplicita

```
*count_p = *count_p + 1;
```

Il risultato netto è quindi quello di aumentare di `1` il valore della variabile `count` ogni volta che la funzione aggiunge un nuovo valore alla somma, come mostrato dall'output del programma:

```
var:   1 -> somma:   1 -> count:   1
var:  -2 -> somma:   1 -> count:   1
var:  -3 -> somma:   1 -> count:   1
var:   4 -> somma:   5 -> count:   2
var:   5 -> somma:  10 -> count:   3
var:   6 -> somma:  16 -> count:   4
var:   7 -> somma:  23 -> count:   5
```

Osserviamo che se fosse stato utilizzato l'operatore di incremento "`++`" in forma postfissa era necessario utilizzare le parentesi poiché l'istruzione

```
*count_p++;
```

non aumenta di `1` il valore della variabile puntata da `count_p` ma restituisce il valore della variabile puntata da `count_p` ed sposta il puntatore `count_p` all'unità di memoria che si trova `sizeof(int)` posizioni dopo quella puntata da `count_p`. Infatti poiché l'operatore di incremento "`++`" ha un livello di precedenza più alto dell'operatore di dereferenza "`*`" l'istruzione precedente viene letta come

```
*(count_p++);
```

per cui, siccome il valore dell'operatore "`++`" in forma postfissa è il valore dell'operando prima dell'incremeneto, viene aumentato di `1` il valore del puntatore `count_p`, spostandosi quindi di `sizeof(int)` posizioni in avanti, e poi restituito il valore contenuto nella locazione di memoria puntata da `count_p` prima dell'incremento. Per "forzare" l'interpretazione corretta bisogna utilizzare le parentesi

```
(*count_p)++;
```

in modo che venga effettuata prima l'operazione di dereferenza e poi quella di incremento. La precedente istruzione è infatti equivalente a

```
*count_p = *count_p + 1;
```

come nel caso precedente.

Nel caso dell'operatore di incremento in forma prefissa utilizzato nella funzione `sum_var()` le parentesi non sono necessarie poiché l'associatività dell'operatore "`++`" è a destra. Di conseguenza nonostante l'operatore "`++`" abbia la precedenza sull'operatore "`*`" la sua associatività forza la valutazione di `*count_p`, che si trova alla sua destra, prima dell'operazione di incremento, fornendo così il risultato voluto. Le parentesi possono tuttavia essere usate

ugualmente, e se ne consiglia l'uso ogni qual volta vi siano dubbi sulla corretta interpretazione di una istruzione,

```
++(*count_p);
```

per aumentare la chiarezza dell'istruzione.

La differenza tra la forma `++(*count_p)` e la forma `(*count_p)++` è nel valore dell'espressione che nel primo caso con l'operatore "`++`" in forma prefissa è il valore della variabile puntata da `count_p` dopo l'incremenento, mentre nel secondo caso con l'operatore "`++`" in forma postfissa è il valore della variabile puntata da `count_p` prima dell'incremento. Tuttavia siccome nella funzione `sum_var()` il valore dell'espressione non viene usato le due espressioni `++(*count_p)` e `(*count_p)++` sono perfettamente equivalenti.

Anche il valore della variabile `sum` può essere modificato all'interno della funzione, come mostrato dal seguente programma.

Programma: `sum_var3.c`

```
#include <stdio.h>

void sum_var(int value, int *sum_p, int *count_p);

int main(void)
{
  int i;
  int a[ ] = { 1, -2, -3, 4, 5, 6, 7};
  int sum, count;

  sum   = 0;
  count = 0;
  for (i = 0; i < 7; ++i) {
    sum_var(a[i], &sum, &count);
    printf("var: %3d -> somma: %3d -> count: %3d\n", a[i], sum, count);
  }

  return 0;
}

void sum_var(int value, int *sum_p, int *count_p)
{
  if (value > 0) {
    ++*count_p;
    *sum_p += value;
  }
  return;
}
```

In questo caso la funzione `sum_var()` è di tipo `void` non dovendo restituire nessun valore. Ricordiamo che l'istruzione `return` va comunque messa, anche se la funzione non restituisce nessun valore.

Infine il seguente programma mostra come il problema del contatore possa essere risolto

utilizzando una variabile con classe di memorizzazione `static`. In questo caso la funzione `sum_var()` restituisce il numero di variabili sommate.

Programma: `sum_var4.c`

```
#include <stdio.h>

int sum_var(int value, int *sum_p);

int main(void)
{
  int i;
  int a[ ] = { 1, -2, -3, 4, 5, 6, 7};
  int sum, count;

  sum   = 0;
  for (i = 0; i < 7; ++i) {
    count = sum_var(a[i], &sum);
    printf("var: %3d -> somma: %3d -> count: %3d\n", a[i], sum, count);
  }

  return 0;
}

int sum_var(int value, int *sum_p)
{
  static int count = 0;

  if (value > 0) {
    ++count;
    *sum_p += value;
  }
  return count;
}
```

La variabile locale `count` della funzione `sum_var()` è dichiarata con classe di memorizzazione `static` per cui viene inizializzata con il valore 0 solo una volta quando viene creata e non distrutta quando l'esecuzione della funzione termina mantenendo così il suo valore da una chiamata all'altra della funzione. Ricordiamo che le variabili statiche sono inizializzate a zero per *default* per cui in questo caso l'inizializzazione esplicita poteva essere omessa. Tuttavia per una maggiore chiarezza del programma si consiglia di utilizzare sempre inizializzazioni esplicite, anche se superflue.

Utilizzando i puntatori siamo ora in grado di scrivere correttamente la funzione `swap()` per scambiare il valore di due variabili:

```
void swap(double *x, double *y)
{
  double temp;
  temp  = *x;
  *x    = *y;
  *y    = temp;
```

```
    return;
}
```

Adesso l'istruzione

```
swap(&a,&b);
```

scambia i valori contenuti nelle variabili `a` e `b`.

Parametri di tipo `void *`

Nel linguaggio C è possibile definire funzioni con parametri formali puntatori ad oggetti di tipo `T` generico utilizzando il tipo `void *`. In altre parole un parametro formale dichiarato di tipo `void *` nel prototipo della funzione può contenere l'indirizzo di memoria di un oggetto di tipo `T` qualsiasi.

Il seguente programma mostra un semplice esempio

Programma: `func_void_par.c`

```c
#include <stdio.h>

double f(void *x);

int main(void)
{
  char       c;
  int        i;
  double a[10];

  f(&c);                /* Cast esplicito non necessario */
  f(&i);                /* Cast esplicito non necessario */
  f(a);                 /* Cast esplicito non necessario */
  f((void *) f);        /* Cast esplicito necessario     */

  return 0;
}

double f(void *x)
{
  printf("indirizzo di memoria: %p\n", x);
  return 0;
}
```

Osserviamo che nel caso di variabili o arrays il cast esplicito a tipo `void *` del valore dell'argomento della funzione non è necessario in quanto la conversione viene fatta automaticamente quando il valore del puntatore all'oggetto viene assegnato al parametro della funzione di tipo `void *`. Nel caso di puntatore a funzione invece il cast esplicito è necessario perché nello Standard C non è permesso assegnare il valore di un puntatore a funzione ad un puntatore di tipo `void *`.

Un altro esempio di funzioni che prendono parametri di tipo `void *` sono le funzioni di output `printf()`, `fprintf()` e `sprintf()` in corrispondenza della direttiva di conversione "`%p`" per stampare il valore di un puntatore.

2.26. Puntatori ed arrays (Rev. 2.1.2)

Nel linguaggio C vi è una stretta corrispondenza tra gli arrays di tipo `T` e i puntatori a tipo `T` perché ogni qual volta un identificatore di un array compare in un'espressione questo viene automaticamente convertito da tipo "array di tipo `T`" a tipo "puntatore a tipo `T`" e gli viene assegnato come valore l'indirizzo di memoria del primo elemento dell'array. Ad esempio se definiamo

```
int array[10];
int  *array_p;
```

allora le due istruzioni

```
array_p = &array[0];
```

e

```
array_p = array;
```

sono perfettamente equivalenti perché l'identificatore `array` viene automaticamente convertito in `&array[0]` prima di effettuare l'assegnazione.

L'unica eccezione a questa regola è quando l'identificatore dell'array compare come operando dell'operatore `sizeof`. In questo caso infatti l'identificatore non viene convertito cosicché l'operatore `sizeof` restituisce correttamente la dimensione dell'intera array e non quella del puntatore al primo elemento dell'array. Nel precedente esempio il valore di `sizeof(array)` è quindi `10*sizeof(int)` e non `sizeof(int *)` come può essere facilmente verificato con il seguente semplice programma:

```
#include <stdio.h>

int main(void)
{
  int array[10];
  printf("%d %d %d\n", sizeof(array), 10*sizeof(int),
                       sizeof(int *));
  return 0;
}
```

Gli identificatori degli arrays sono convertiti a puntatori costanti in modo che il loro valore non possa essere cambiato e che quindi puntino sempre al primo elemento dell'array. Di conseguenza se `array` è l'identificatore di un array espressioni del tipo

```
++array   array += 2   array = array_p
```

sono illegali.

Non solo gli identificatori degli arrays sono convertiti in puntatori, ma poiché l'espressione `array[i]` viene definita nel linguaggio C come

$$\texttt{array[i]} \Longleftrightarrow \texttt{*(array + i)}$$

dove `array` è convertita in `&array[0]`, tutte le operazioni di indice degli arrays sono definite in termini dell'aritmetica dei puntatori. Il seguente programma mostra questa equivalenza.

Programma: `array_vs_pointers.c`

```
#include <stdio.h>

int main(void)
{
  int ind;
  int array[10];

  for (ind = 0; ind < 10; ++ind) array[ind] = ind;

  printf ("ind    array[ind]    *(array+ind) \t &array[ind] \t "
          "(array+ind)\n\n");
  for (ind = 0; ind < 10; ind++)
    {
      printf("%2d\t %2d\t\t  %2d\t\t %-10p\t %-10p\t \n",
             ind,
             array[ind], *(array + ind),
             (void *) &array[ind], (void *) (array + ind)
             );
    }

  return 0;
}
```

Ricordiamo che la direttiva di conversione "`%p`" si aspetta un puntatore di tipo `void *` di conseguenza viene effettuato un cast esplicito del valore degli argomenti della funzione `printf()`:

```
printf("%2d\t %2d\t\t  %2d\t\t %-10p\t %-10p\t \n",
       ind,
       array[ind], *(array + ind),
       (void *) &array[ind], (void *) (array + ind)
       );
```

Se questo non viene fatto il compilatore può produrre un messaggio di *warning*.

Osserviamo che come conseguenza dell'identificazione di `array[ind]` con `*(array+ind)` si ha che `array[ind]` è equivalente a `ind[array]`, per cui al posto di

```
for(ind = 0; ind < 10; ++ind) array[ind] = ind;
```

nel programma si sarebbe potuto utilizzare

```
for(ind = 0; ind < 10; ++ind) ind[array] = ind;
```

anche se la prima forma può risultare più chiara.

Quando il programma viene eseguito si ha un output del tipo

ind	array[ind]	*(array+ind)	&array[ind]	(array+ind)
0	0	0	0xbffffb80	0xbffffb80
1	1	1	0xbffffb84	0xbffffb84
2	2	2	0xbffffb88	0xbffffb88
3	3	3	0xbffffb8c	0xbffffb8c
4	4	4	0xbffffb90	0xbffffb90
5	5	5	0xbffffb94	0xbffffb94
6	6	6	0xbffffb98	0xbffffb98
7	7	7	0xbffffb9c	0xbffffb9c
8	8	8	0xbffffba0	0xbffffba0
9	9	9	0xbffffba4	0xbffffba4

Il valore degli indirizzi di memoria assegnati dipendono chiaramente dal sistema tuttavia indipendentemente dal loro valore ogni volta che l'indice `ind` viene incrementato di `1` il valore del puntatore aumenta della dimensione del tipo `int`, `4` su questo sistema, cosicché il puntatore punta correttamente all'elemento successivo dell'array, e non all'unità di memoria successiva. Questo risultato illustra chiaramente la differenza fondamentale che vi è tra l'aritmetica dei puntatori e quella dei numeri interi già notata in precedenza quando sono state introdotte le operazioni con puntatori. Quando un numero intero `n` viene aggiunto o sottratto ad un puntatore a tipo `T` il risultato dell'operazione è quella di muoversi in avanti o indietro nella memoria di `n` locazioni di memoria ciascuna di dimensione `sizeof(T)`, ossia composta di `sizeof(T)` unità di memoria. Questo vuol dire che il valore del puntatore non varia di `n` ma bensì di `n*sizeof(T)`, come illustrato dal seguente programma

Programma: `pointer_arit.c`

```
#include <stdio.h>

int main(void)
{
  int    a[2];
  double b[2];

  printf("sizeof(int)   : %u\n", (unsigned int) sizeof(int));
  printf("(a+1) - a     : %d\n",  (a+1) -  a);
  printf("size(a+1 - a) : %d\n",  (int) (a+1) - (int) a);
  printf("\n");
  printf("sizeof(double): %u\n", (unsigned int) sizeof(double));
  printf("(b+1) - b     : %d\n",  (b+1) -  b);
  printf("size(b+1 - b) : %d\n",  (int) (b+1) - (int) b);


  return 0;
}
```

Il tipo dell'operatore `sizeof` può dipendere dal sistema di conseguenza per eliminare l'eventuale inconsistenza tra la direttiva di conversione "`%u`" ed il tipo dell'operatore `sizeof` viene effettuato un cast esplicito del valore restituito dall'operatore al tipo `unisgned int`.

L'output del programma dipende dalla dimensione dei tipi `int` e `double`, se queste sono ad esempio 4 e 8 si ha

```
sizeof(int)    : 4
(a+1) - a      : 1
size(a+1 - a)  : 4

sizeof(double): 8
(b+1) - b      : 1
size(b+1 - b)  : 8
```

Dal momento che le operazioni di indice degli arrays sono definite in termini dell'aritmetica dei puntatori l'indice può essere utilizzato direttamente con i puntatori, e non solo con gli identificatori degli arrays. Ad esempio se `p` è un puntatore a tipo `T` l'espressione `p[i]`, ovvero `*(p+i)`, è il contenuto della locazione di memoria di dimensione `sizeof(T)` che si trova `i*sizeof(T)` unità di memoria dopo l'unità di memoria puntata da `p`. Naturalmente è cura del programmatore far si che il puntatore `p` punti ad un array di elementi e dimensioni appropriati, come nell'esempio seguente

```
int a[10], i;
int *p;
p = a;
for(i = 0; i < 10; ++i) p[i] = i;
```

In caso contrario il risultato potrebbe essere disastroso.

Per illustrare l'uso dei puntatori con gli arrays supponiamo di volere scrivere un programma che conti quanti sono gli elementi consecutivi, a partire dal primo elemento, diversi da 0 di un array di tipo `int`. Questo si ottiene facilmente come

Programma: `array_length.c`

```
#include <stdio.h>

int main(void)
{
  int i;
  int array[ ] = {1, 2, 3, 4, 0, 5, 6, 7, 8, 9};

  i = 0;
  while ( array[i] != 0) ++i;

  printf("numero di elementi prima di 0: %d\n", i);

  return 0;
}
```

Lo stesso risultato si può ottenere utilizzando per muoversi lungo l'array un puntatore al posto dell'indice:

Programma: `array_length1.c`

```c
#include <stdio.h>

int main(void)
{
  int *array_p;
  int  array[ ] = {1, 2, 3, 4, 0, 5, 6, 7, 8, 9};

  array_p = array;

  while ( (*array_p) != '\0') ++array_p;

  printf("numero di elementi prima di 0: %d\n", array_p - array);

  return 0;
}
```

Sfruttando le proprietà dell'aritmetica dei puntatori il numero di elementi consecutivi diversi da 0 a partire dal primo elemento dell'array è dato dalla differenza tra il valore del puntatore `array_p`, che contiene l'indirizzo di memoria dell'ultimo elemento dell'array prima dello 0, e quello di `array` il cui valore è l'indirizzo di memoria del primo elemento dell'array.

Sebbene il risultato dei due programmi sia lo stesso il secondo è più efficiente perché `array[i]` è definito come `*(array+i)` e quindi comporta un'operazione in più rispetto alla dereferenza `*array_p` utilizzata nel secondo programma.

Osserviamo infine che entrambi i programmi assumono che l'array contenga almeno uno zero. Se questo non accade il ciclo `while` non finisce mai. Quando un programma fallisce perché non vengono soddisfatte le sue richieste, ma non viene segnalato nulla, si dice che il programma non è *fail safe*.

2.26.1. Puntatori, arrays e puntatori a tipo array

Abbiamo visto che la dichiarazione

```c
type (*array_ptr_name)[size]
```

dichiara il puntatore `array_ptr_name` ad array di `size` elementi di tipo `type`. Ad esempio la seguente istruzione

```c
int (*a_p)[10];
```

dichiara il puntatore `a_p` ad array di `10` elementi di tipo `int`. Di conseguenza se viene definita un array `a` come

```c
int a[10];
```

l'istruzione

```c
a_p = &a;
```

assegna al puntatore `a_p` l'indirizzo di memoria dell'array `a`, cosicché l'espressione `(*a_p)[i]` fornisce il valore dell'elemento `i` dell'array `a`. Ricordiamo che le parentesi "`()`" sono necessarie.

Trasformando l'espressione `(*a_p)[i]` secondo la definizione dell'operazione di indice "`[]`" si ottiene

$$\texttt{(*a\_p)[i]} \Longleftrightarrow \texttt{*(*a\_p + i)}$$

In realtà secondo la definizione si avrebbe `*((*a_p) + i)`, tuttavia le parentesi più interne sono irrilevanti e possono essere omesse per semplicità di scrittura. Le parentesi più esterne invece sono necessarie affinché l'espressione sia valutata correttamente:

`*a_p`	⇒	indirizzo primo elemento di `a`
`*a_p + i`	⇒	indirizzo elemento `i` di `a`
`*(*a_p + i)`	⇒	valore elemento `i` di `a`

La precedente scrittura mostra chiaramente che per accedere al valore dell'elemento di indice `i` dell'array `a` a partire dal puntatore all'array `a_p` sono necessarie due operazioni di dereferenza. Questo non dovrebbe sorprendere più di tanto perché essendo l'identificatore `a` dell'array convertito in puntatore al primo elemento dell'array l'istruzione

```
a_p = &a;
```

assegna al puntatore `a_p` l'indirizzo di memoria di un puntatore. Il puntatore `a_p` è quindi un puntatore a puntatore. In altre parole il puntatore `a_p` contiene l'indirizzo di memoria di una locazione di memoria che a sua volta contiene l'indirizzo di memoria del primo elemento dell'array. Di conseguenza per accedere al valore degli elementi dell'array dal puntatore all'array sono necessarie due operazioni di dereferenza: la prima per conoscere l'indirizzo di memoria del primo elemento dell'array, la seconda per conoscere il valore dell'elemento voluto a partire dall'indirizzo di memoria del primo.

Il seguente programma illustra questo punto.

Programma: `pointer_array2.c`

```
#include <stdio.h>

int main(void)
{
  int           i;
  double     a[5];
  double (*a_p)[5];

  a_p = &a;
  for (i = 0; i < 5; ++i) a[i] = 2.0 * i + 1.0;

  printf("  i    a[i]     *(*a_p+i)\n");
  for (i = 0; i < 5; ++i)
    printf("%3d -> %.3f     %.3f\n", i, a[i], *(*a_p+i));

  printf("sizeof(a_p)     : %3lu   sizeof(a)     : %3lu",
         sizeof(a_p), sizeof(a));
  printf("   sizeof(&a)     : %3lu\n", sizeof(&a));
```

```
  printf("sizeof(double *): %3lu   sizeof(double): %3lu\n",
         sizeof(double *), sizeof(double));

  return 0;
}
```

Quando il programma viene eseguito si ha il seguente output

```
    i     a[i]      *(*a_p+i)
    0 -> 1.000       1.000
    1 -> 3.000       3.000
    2 -> 5.000       5.000
    3 -> 7.000       7.000
    4 -> 9.000       9.000
  sizeof(a_p)      :   4   sizeof(a)      :  40   sizeof(&a)      :   4
  sizeof(double *):   4   sizeof(double):   8
```

Osserviamo che la dimensione di `a` è la dimensione dell'array `5*sizeof(double)` mentre quella di `a_p`, come quella di `&a`, è quella di un puntatore a `double`.

Puntatori, arrays ed operatori di incremento/decremento

Quando si usano i puntatori con gli arrays si fa spesso uso degli operatori di incremento "`++`" e decremento "`--`". Questi operatori hanno la precedenza sugli operatori di dereferenza "`*`" e referenza "`&`" ed inoltre il loro valore è differente a seconda che l'operatore compaia in forma prefissa o postfissa. Di conseguenza l'uso prefisso o postfisso può non essere equivalente, come mostrato nel seguente frammento di programma:

```
int array[5];
int  var;
int *var_p;

var_p = &array[0];    /* var_p -> array[0]                    */

var = *var_p++;       /* var   = *var_p     = array[0]        */
                      /* var_p =  var_p + 1 = array[1]        */

var = *++var_p;       /* var   = *(var_p + 1) = array[2]      */
                      /* var_p =  var_p + 1   = array[2]      */

var = ++*var_p;       /*  var   = (*var_p + 1) = array[2] + 1 */
                      /* *var_p = *var_p + 1                  */
                      /*     ossia array[2] = array[2] + 1    */
                      /* var_p inalterato                     */
```

Questa differenza di interpretazione è spesso causa di errori di difficile individuazione.

2.27. Arrays come parametri di funzione (Rev. 2.1)

L'utilizzo di arrays come parametri di funzione non differisce molto dall'uso di puntatori come parametri di funzione. Infatti poiché il linguaggio C converte automaticamente nelle espressioni gli identificatori degli arrays in puntatori al primo elemento dell'array una chiamata di funzione del tipo

```
function(array)
```

con `array` un array di tipo `T` qualunque è a tutti gli effetti equivalente alla chiamata

```
function(&array[0])
```

per cui alla funzione `function()` viene passato il puntatore al primo elemento dell'array. Conseguenza diretta di ciò è che gli elementi di un array fornito come parametro ad una funzione possono essere modificati dalla funzione.

Una conversione analoga avviene anche nei prototipi delle funzioni, sia che compaiano nella definizione che nella dichiarazione di funzione. In altre parole un parametro formale del prototipo dichiarato di tipo "array di tipo `T`", completo o incompleto, viene automaticamente convertito in un parametro formale di tipo "puntatore a tipo `T`". Ad esempio le dichiarazioni seguenti della funzione `function()`

```
int function(int array[10]);    /* array tipo int completo   */
int function(int array[ ]);     /* array tipo int incompleto */
int function(int *array);       /* puntatore a tipo int      */
```

sono tutti equivalenti tra loro. Inoltre dal momento che i nomi dei parametri formali nei prototipi delle dichiarazioni di funzione non sono necessari la funzione `function()` può anche essere dichiarata come

```
int function(int [10]);         /* array tipo int completo   */
int function(int [ ]);          /* array tipo int incompleto */
int function(int *);            /* puntatore a tipo int      */
```

risparmiando in scrittura ma perdendo molto probabilmente in chiarezza.

Il seguente programma mostra un semplice esempio di utilizzo di arrays con le funzioni.

Programma: `func_array.c`

```
#include <stdio.h>

void init_array(double array[], int size);

int main(void)
{
  int      ind;
  double a[10];

  init_array(a, 10);

  for (ind = 0; ind < 10; ++ind)
```

```
    printf("array[%1d] = %.3f\n", ind, a[ind]);

  return 0;
}

void init_array(double *array, int size)
{
  int ind;

  for (ind = 0; ind < size; ++ind)
    array[ind] = (double) ind;

  return;
}
```

In questo esempio nel prototipo della dichiarazione di funzione ed in quello della definizione di funzione sono state utilizzate *volutamente* due dichiarazioni diverse dei parametri formali per mostrarne la completa equivalenza.

Osserviamo inoltre che essendo il primo parametro della funzione `init_array()` un puntatore a tipo `double` al posto dell'istruzione

```
array[ind] = (double) ind;
```

si può utilizzare l'istruzione

```
*(array + ind) = (double) ind;
```

ovvero, essendo l'operazione di indice `array[ind]` definita come `*(array+ind)`, l'istruzione meno usuale

```
ind[array] = (double) ind;
```

Tutte e tre queste istruzioni infatti assegnano il valore della variabile `ind`, convertito in tipo `double`, all'elemento di indice `ind` dell'array di tipo `double` il cui primo elemento si trova all'indirizzo di memoria indicato dal puntatore `array`.

La chiamata a funzione nella funzione `main()` è stata effettuata specificando solo l'identificatore dell'array `a`

```
init_array(a, 10);
```

tuttavia è possibile utilizzare anche la forma equivalente

```
init_array(&a[0], 10);
```

per indicare esplicitamente che il primo argomento della funzione contiene l'indirizzo di memoria del primo elemento dell'array `a`. Usualmente, e a meno che non sia necessario per altri motivi, nella chiamata a funzioni con arrays si utilizza la prima forma in cui compare solo l'identificatore degli arrays.

Le chiamate seguenti sono invece errate in questo contesto

```
init_array(a[0], 10);   /* passa primo elemento array */
init_array(&a, 10);     /* passa puntatore ad array   */
```

Infatti la prima chiamata passa alla funzione il valore del primo elemento dell'array `a`, e non il suo indirizzo di memoria. La seconda chiamata invece passa l'indirizzo di memoria del puntatore al primo elemento dell'array `a`, ossia il puntatore all'array che è di tipo `double (*)[]` e non di tipo `double *` come richiesto dalla funzione.

Osserviamo che per come è scritta la funzione `init_array()` la chiamata

```
init_array(&a[2], 8);
```

è perfettamente lecita ed ha come risultato che solo gli elementi dell'array `a` dal terzo all'ultimo vengono modificati.

Nota di stile. Quando si scrive un programma si deve cercare di renderlo il più possibile chiaro e leggibile. Un programma scritto in modo molto criptico è un chiaro esempio di cattiva programmazione. I puntatori si prestano particolarmente bene ad una scrittura di difficile interpretazione come mostrato dalla seguente funzione che copia una stringa su un'altra:

```
void string_copy(char *p, char *q)
{
  while (*p++ = *q++);
  return;
}
```

La stessa funzione può essere scritta in modo molto più trasparente come

```
void copy_string(char *source, char *dest)
{
  while (1) {
    *dest = *source;

    if (*dest == '\0') return;  /* se fine stringa esci */
    ++dest;
    ++source;
  }
}
```

Questa forma è chiaramente da preferirsi alla precedente.

2.28. Arrays multidimensionali, puntatori e funzioni (Rev. 2.1)

Il linguaggio C permette di definire arrays di dimensione qualsiasi semplicemente aggiungendo "dimensioni" nella dichiarazione. Ad esempio le dichiarazioni

```
int arr_1[7];                /* Array                    */
int arr_2[3][4];             /* Array a due dimensioni   */
int arr_3[3][4][5];          /* Array a tre dimensioni   */
```

dichiarano l'array `arr_1` di dimensione 7 di tipo `int`, l'array bidimensionale `arr_2` di dimensione 3×4 di tipo `int` e l'array tridimensionale di dimensione $3 \times 4 \times 5$ di tipo `int`.

2.28.1. Organizzazione in memoria degli arrays multidimensionali

Gli elementi degli arrays multidimensionali sono ordinati in memoria in modo tale che gli elementi il cui ultimo indice a destra differisce di una unità occupano locazioni di memoria contigue, come mostrato dal seguente programma per un array bidimensionale:

Programma: `arr_2-addr.c`

```
#include <stdio.h>

int main(void)
{
  int           i, j;
  double arr_2[3][4];

  printf("\nElemento\t   \t  Indirizzo\n\n");
  for (i = 0; i < 3; ++i) {
    for (j = 0; j < 4; ++j) {
      printf("arr_2[%1d][%1d]\t-->\t  %-10p  (%lu)\n",
             i, j,
             (void *) &arr_2[i][j],
             (unsigned long) &arr_2[i][j]
             );
    }
  }
  printf("\nsizeof(double) = %u\n", sizeof(double));

  return 0;
}
```

Il programma assume che l'operatore `sizeof` restituisca un valore di tipo `unsigned int`.[8] Nel caso il sistema utilizzi un tipo diverso il compilatore può produrre un messaggio di *warning* per segnalare l'inconsistenza tra la direttiva di conversione "`%u`" ed il valore restituito dall'operatore. In questo caso è sufficiente cambiare la direttiva di conversione o, alternativamente, effettuare un cast esplicito del valore restituito dall'operatore `sizeof` al tipo utilizzato nella direttiva di conversione

```
printf("\nsizeof(int) = %u\n", (unsigned int) sizeof(int));
```

Come per tutti i programmi che stampano il valore di indirizzi di memoria l'output dipende ovviamente dal sistema, sul mio ha prodotto

```
Elemento                    Indirizzo

arr_2[0][0]      -->        0xbffffad0  (3221224144)
arr_2[0][1]      -->        0xbffffad8  (3221224152)
arr_2[0][2]      -->        0xbffffae0  (3221224160)
arr_2[0][3]      -->        0xbffffae8  (3221224168)
arr_2[1][0]      -->        0xbffffaf0  (3221224176)
```

[8] L'operatore `sizeof` restituisce un valore di tipo `size_t` che a seconda del sistema è generalmente `unsigned int` o `unsigned long int`.

```
arr_2[1][1]     --->     0xbffffaf8  (3221224184)
arr_2[1][2]     --->     0xbffffb00  (3221224192)
arr_2[1][3]     --->     0xbffffb08  (3221224200)
arr_2[2][0]     --->     0xbffffb10  (3221224208)
arr_2[2][1]     --->     0xbffffb18  (3221224216)
arr_2[2][2]     --->     0xbffffb20  (3221224224)
arr_2[2][3]     --->     0xbffffb28  (3221224232)

sizeof(double) = 8
```

È facile tuttavia convincersi che indipendentemente dai valori riportati nelle ultime due colonne l'indirizzo di memoria dell'elemento `arr_2[i][j]` dell'array bidimensionale è dato dall'indirizzo di memoria del primo elemento dell'array più il valore del primo indice `i` per la seconda dimensione dell'array, 4, più il valore del secondo indice:

```
&arr_2[0][0] + 4 * i + j
```

Ricordiamo che se si aumenta di uno il valore di un puntatore a tipo `T` il valore dell'indirizzo di memoria aumenta di `sizeof(T)`, ossia di `sizeof(double) = 8` nel nostro caso.

Se rappresentiamo l'array bidimensionale `arr_2` come una matrice di 3 righe e 4 colonne

	col 1	**col 2**	**col 3**	**col 4**
riga 1	`arr_2[0][0]`	`arr_2[0][1]`	`arr_2[0][2]`	`arr_2[0][3]`
riga 2	`arr_2[1][0]`	`arr_2[1][1]`	`arr_2[1][2]`	`arr_2[1][3]`
riga 3	`arr_2[2][0]`	`arr_2[2][1]`	`arr_2[2][2]`	`arr_2[2][3]`

si vede facilmente gli che elementi degli arrays bidimensionali sono ordinati in memoria per *righe*.

L'ordinamento degli elementi degli arrays di dimensione superiore a due segue una logica del tutto analoga. Ad esempio l'elemento `arr_3[i][j][k]` dell'array tridimensionale

```
double arr_3[2][3][4];
```

occupa la locazione di memoria di indirizzo

```
&arr_3[0][0][0] + 3 * 4 * i + 4 * j + k
```

come mostra il seguente programma

Programma: `arr_3-addr.c`

```
#include <stdio.h>

int main(void)
{
  int    i, j, k;
  double arr_3[2][3][4];

  printf("\nElemento\t    \t  Indirizzo\n\n");
  for (i = 0; i < 2; ++i) {
```

```
    for (j = 0; j < 3; ++j) {
      for (k = 0; k < 4; ++k) {
        printf("arr_3[%1d][%1d][%1d]\t-->\t  %-10p  %-10p\n",
               i, j, k,
               (void *) &arr_3[i][j][k],
               (void *) (&arr_3[0][0][0] + 3 * 4 * i + 4 * j + k)
               );
      }
    }
  }
  printf("\nsizeof(double) = %u\n", sizeof(double));

  return 0;
}
```

il cui output (parziale) è

```
Elemento                    Indirizzo

arr_3[0][0][0]  -->         0xbffffa70  0xbffffa70
arr_3[0][0][1]  -->         0xbffffa78  0xbffffa78
arr_3[0][0][2]  -->         0xbffffa80  0xbffffa80
arr_3[0][0][3]  -->         0xbffffa88  0xbffffa88
arr_3[0][1][0]  -->         0xbffffa90  0xbffffa90
arr_3[0][1][1]  -->         0xbffffa98  0xbffffa98
arr_3[0][1][2]  -->         0xbffffaa0  0xbffffaa0
arr_3[0][1][3]  -->         0xbffffaa8  0xbffffaa8
arr_3[0][2][0]  -->         0xbffffab0  0xbffffab0
arr_3[0][2][1]  -->         0xbffffab8  0xbffffab8
....
arr_3[1][1][3]  -->         0xbffffb08  0xbffffb08
arr_3[1][2][0]  -->         0xbffffb10  0xbffffb10
arr_3[1][2][1]  -->         0xbffffb18  0xbffffb18
arr_3[1][2][2]  -->         0xbffffb20  0xbffffb20
arr_3[1][2][3]  -->         0xbffffb28  0xbffffb28

sizeof(double) = 8
```

La visualizzazione degli arrays di dimensione superiore a due in termini di matrici è meno immediata ma non complicata. Come sarà chiaro tra breve ad esempio un array tridimensionale può essere visto come un array i cui elementi sono arrays bidimensionali.

2.28.2. Identificatori degli arrays multidimensionali e puntatori

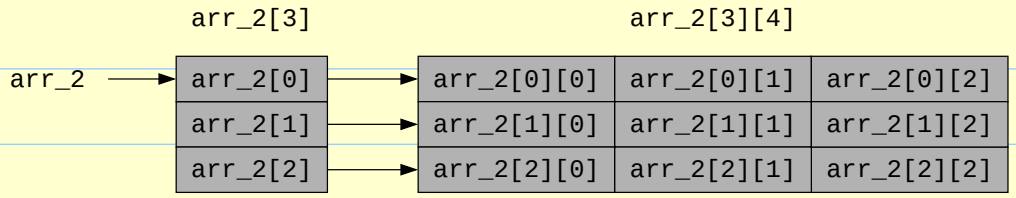
Come avviene per identificatori degli arrays anche gli identificatori degli arrays multidimensionali sono convertiti in puntatori, la conversione è tuttavia più complessa a causa della presenza di più di un operatore di indice "[]". Il modo migliore per comprendere come un identificatore di un array multidimensionale viene convertito in puntatore è con un esempio. Consideriamo la dichiarazione

```
int arr_2[3][4];
```

che dichiara l'array bidimensionale `arr_2`. Nella dichiarazione vi sono due operatori di indice che avendo la stessa priorità vengono quindi applicati nell'ordine specificato dalla loro associatività. Siccome l'associatività dell'operatore di indice "`[]`" è a sinistra questo vuol dire che ciascun operatore "`[]`" si applica a tutto quello che vi è alla sua sinistra con il risultato che la dichiarazione precedente viene interpretata come

```
int (arr_2[3])[4];
```

Di conseguenza l'array bidimensionale `arr_2` viene visto come un array composto da 3 elementi di tipo array di dimensione 4 di tipo `int`. Applicando ora l'usuale conversione di un identificatore di un array di tipo `T` in puntatore a tipo `T` segue che l'identificatore `arr_2` dell'array bidimensionale viene convertito in "puntatore a tipo array di dimensione 4 di tipo `int`" ed il suo valore è l'indirizzo di memoria del primo elemento dell'array `arr_2[3]` di dimensione 3 i cui elementi sono identificatori di arrays di dimensione 4 di tipo `int`. Di conseguenza ciascun elemento dell'array `arr_2[3]` è un puntatore il cui valore è l'indirizzo di memoria del primo elemento degli arrays di dimensione 4 di tipo `int` che rappresentano le righe della matrice associata. La seguente figura illustra schematicamente questa organizzazione.



Per comodità l'array bidimensionale `arr_2[3][4]` è stata rappresentata con la matrice 3×4 associata e non secondo l'ordinamento reale nella memoria.

Il procedimento può essere ripetuto per arrays con un numero qualsiasi di indici per cui in generale ogni espressione del tipo "array di dimensione $i \times j \times \cdots \times k$ di tipo `T`" viene automaticamente convertita in "puntatore a tipo array di dimensione $j \times \cdots \times k$ di tipo `T`".

Conseguenza di ciò è che gli arrays multidimensionali nel linguaggio C sono di fatto "arrays di arrays" e tutte le operazioni di indice sono definite anche per gli arrays multidimensionali attraverso l'aritmetica dei puntatori. Ad esempio l'espressione `arr_2[1][2]` viene definita come

```
arr_2[1][2]  ⇒  (arr_2[1])[2]
             ⇒  (*(arr_2 + 1))[2]
             ⇒  *(*(arr_2 + 1) + 2)
```

infatti

- `arr_2`
  puntatore al primo elemento dell'array di dimensione 3 di tipo array di dimensione 4 di tipo `int`.
- `arr_2 + 1`
  puntatore al secondo elemento dell'array di dimensione 3 di tipo array di dimensione 4

di tipo `int`.

- `*(arr_2 + 1) = arr_2[1]`
  array di dimensione 4 di tipo `int` secondo elemento dell'array di dimensione 3 di tipo array di dimensione 4 di tipo `int`: seconda riga della matrice. Il valore è il puntatore al primo elemento dell'array di dimensione 4 di tipo `int`.

- `*(arr_2 + 1) + 2 = arr_2[1] + 2`
  puntatore al terzo elemento dell'array di dimensione 4 di tipo `int` secondo elemento dell'array di dimensione 3 di tipo array di dimensione 4 di tipo `int`: terzo elemento della seconda riga della matrice.

- `*(*(arr_2 + 1) + 2) = *(arr_2[1] + 2)= (arr_2[1])[2] = arr_2[1][2]`
  valore del terzo elemento dell'array di dimensione 4 di tipo `int` secondo elemento dell'array di dimensione 3 di tipo array di dimensione 4 di tipo `int`: valore del terzo elemento della seconda riga della matrice.

Il seguente programma mostra questa struttura per l'array `arr_2`:

Programma: `arr_2-addr2.c`

```
#include <stdio.h>

int main(void)
{
  int        i, j;
  int arr_2[3][4];

  printf("\nElemento\t   \t  Indirizzo\n");
  for (i = 0; i < 3; ++i) {
      printf ("\narray[%d]\t==> \t  %lu  (%lu)\n\n",
               i,
               (unsigned long) arr_2[i],
               (unsigned long) *(arr_2 + i)
               );

    for (j = 0; j < 4; ++j) {
      printf("arr_2[%1d][%1d]\t-->\t  %lu  (%lu)\n",
             i, j,
             (unsigned long) (arr_2[i] + j),
             (unsigned long) &arr_2[i][j]
             );
    }
  }
  printf("\nsizeof(int) = %u\n", sizeof(int));

  return 0;
}
```

Quando il programma viene eseguito sul mio computer produce il seguente output.

```
Elemento                    Indirizzo

array[0]          ==>       3221224192  (3221224192)

arr_2[0][0]       -->       3221224192  (3221224192)
arr_2[0][1]       -->       3221224196  (3221224196)
arr_2[0][2]       -->       3221224200  (3221224200)
arr_2[0][3]       -->       3221224204  (3221224204)

array[1]          ==>       3221224208  (3221224208)

arr_2[1][0]       -->       3221224208  (3221224208)
arr_2[1][1]       -->       3221224212  (3221224212)
arr_2[1][2]       -->       3221224216  (3221224216)
arr_2[1][3]       -->       3221224220  (3221224220)

array[2]          ==>       3221224224  (3221224224)

arr_2[2][0]       -->       3221224224  (3221224224)
arr_2[2][1]       -->       3221224228  (3221224228)
arr_2[2][2]       -->       3221224232  (3221224232)
arr_2[2][3]       -->       3221224236  (3221224236)

sizeof(int) = 4
```

Si consiglia di confrontare questo output con lo schema della struttura dell'array `arr_2` dato precedentemente. Gli indirizzi di memoria sono scritti in formato decimale per semplificare il confronto.

Per gli arrays di dimensione superiore valgono regole analoghe così ad esempio l'espressione `arr_3[1][2][3]` viene definita come

$$\texttt{arr\_3[1][2][3]} \iff \texttt{*(*(*(arr\_3 + 1) + 2) + 3)}$$

da cui si nota la comodità di poter disporre di un operatore di indice "`[]`".

2.28.3. Arrays multidimensionali e funzioni

La conversione automatica dell'identificatore di un array multidimensionale in puntatore viene effettuata, in modo del tutto analogo a quello che avviene con gli identificatori degli arrays, anche nella dichiarazione dei parametri formali dei prototipi delle funzioni sia che compaiano nella dichiarazione che nella definizione della funzione.

Il seguente programma illustra l'uso di un array bidimensionale come parametro di funzione.

Programma: `arr_2-func.c`

```
#include <stdio.h>

int sum(int (*array)[4]);
```

```
int main(void)
{
  int        i, j;
  int arr_2[3][4];

  for (i = 0; i < 3; ++i) {
    for (j = 0; j < 4; ++j) {
      arr_2[i][j] = 1;
      printf("arr_2[%d][%d] = %d\n", i, j, arr_2[i][j]);
    }
  }
  printf("\nsomma: %d\n", sum(arr_2));

  return 0;
}

int sum(int array[][4])
{
  int i, j;
  int  sum;

  sum = 0;
  for (i = 0; i < 3; ++i) {
    for (j = 0; j < 4; ++j) {
      sum += array[i][j];
    }
  }
  return sum;
}
```

In questo esempio sono state usate due forme equivalenti per la dichiarazione dei parametri formali. Nel prototipo della dichiarazione della funzione `sum()` si è usata la forma

```
int sum(int (*array)[4]);
```

che dichiara esplicitamente il parametro formale `array` come "puntatore ad array di dimensione 4 di tipo `int`". L'identificatore del parametro formale non è necessario in questo caso per cui si sarebbe potuto utilizzare anche la forma equivalente, ma meno trasparente,

```
int sum(int (*)[4]);
```

Nella prototipo della definizione della funzione, invece, si è usata la dichiarazione incompleta

```
int sum(int array[][4])
```

cosicché il parametro formale viene convertito implicitamente da identificatore di un array bidimensionale a puntatore a tipo array di dimensione 4 di tipo `int`. Le due dichiarazioni sono perfettamente equivalenti fra di loro. In entrambi i casi si sarebbe potuto anche utilizzare la dichiarazione completa

```
int sum(int array[3][4])
```

L'utilizzo di arrays di dimensione superiore a due come parametri di funzione avviene in modo del tutto analogo come mostrato dal seguente programma nel caso di un array tridimensionale.

Programma: `arr_3-func.c`

```
#include <stdio.h>

int sum(int (*array)[4][5]);

int main(void)
{
  int        i, j, k;
  int arr_3[3][4][5];

  for (i = 0; i < 3; ++i) {
    for (j = 0; j < 4; ++j) {
      for (k = 0; k < 5; ++k) {
        arr_3[i][j][k] = 1;
        printf("arr_2[%d][%d][%d] = %d\n", i, j, k, arr_3[i][j][k]);
      }
    }
  }
  printf("\nsomma: %d\n", sum(arr_3));

  return 0;
}

int sum(int array[][4][5])
{
  int i, j, k;
  int     sum;

  sum = 0;
  for (i = 0; i < 3; ++i) {
    for (j = 0; j < 4; ++j) {
      for (k = 0; k < 5; ++k) {
        sum += array[i][j][k];
      }
    }
  }
  return sum;
}
```

Osserviamo che per gli arrays multidimensionali, diversamente da quanto avviene per gli arrays, è necessario specificare nei prototipi delle funzioni il valore di tutte le dimensioni, esclusa eventualmente la prima. Il motivo è che l'identificatore di un array "di dimensione $i \times j \times \cdots \times k$ di tipo `T`" viene automaticamente convertito in "puntatore ad array di dimensione $j \times \cdots \times k$ di tipo `T`", per cui il valore di $j, \ldots, k$ deve essere conosciuto per una corretta interpretazione del puntatore. Questo rende l'uso degli arrays multidimensionali piuttosto scomodo, ed è per questo motivo che gli arrays multidimensionali sono utilizzati raramente in questo modo nei programmi in linguaggio C. Vedremo più avanti un modo più agevole di

definire ed utilizzare arrays multidimensionali.

2.29. Funzioni come parametri di funzione (Rev. 2.1)

Il linguaggio C tratta i puntatori a funzione alla stessa stregua di puntatori ad oggetti per cui è possibile utilizzarli in ogni contesto in cui è ammesso l'uso dei puntatori ad esempio come parametri di funzione, purché utilizzati in modo consistente.

Per illustrare l'uso dei puntatori a funzione come parametri di funzione supponiamo di voler scrivere una funzione `sum_func_square()` che calcoli la seguente somma

$$S = \sum_{k=1}^{N} f(k)^2$$

dove $f(x)$ è una funzione data, ma generica. In questo esempio la funzione $f(x)$ è chiaramente un parametro da dover passare alla funzione `sum_func_square()`, l'altro parametro è il valore di N. Siccome è possibile accedere ad una funzione sia direttamente tramite il suo identificatore che indirettamente tramite un puntatore alla funzione, la funzione $f(x)$ può essere passata facilmente alla funzione `sum_func_square()` tramite il suo indirizzo di memoria. La funzione `sum_func_square()` può quindi essere definita come:

Funzione: `sum_func_square()`

```
double sum_func_square(double (*fun)(double), int n)
{
  int      k;
  double sum;
  double tmp;

  sum = 0.0;
  for (k = 1; k <= n; ++k) {
    tmp = (*fun)((double) k);
    sum += tmp * tmp;
  }

  return sum;
}
```

Per evitare di dover fare due chiamate a funzione per ogni valore della variabile `k` la funzione `sum_func_square()` utilizza la variabile *temporanea* `tmp` su cui memorizzare il valore della funzione per il valore corrente di `k`.

Il primo parametro della funzione è un puntatore a funzione di tipo

```
double func_name(double);
```

per cui la funzione `sum_func_square()` può essere utilizzata per calcolare la somma S con ogni funzione il cui prototipo è di questo tipo.

Nella definizione della funzione si sarebbe potuto utilizzare anche il prototipo

```
double sum_func_square(double (*fun)(), int n)
```

che specifica solo che il primo parametro della funzione `sum_func_square()` è un puntatore a funzione di tipo `double` ma non fornisce informazioni sul numero ed il tipo dei parametri formali della funzione puntata da `fun`. Sebbene questo sia perfettamente lecito questa forma nasconde il fatto che la funzione `sum_func_square()` è scritta per funzioni con un solo parametro di tipo `double`. In generale è una buona norma di programmazione quella di utilizzare dichiarazioni e definizioni di funzione il più esplicite possibile, non solo per la chiarezza del programma ma anche per permettere al compilatore di controllare che le funzioni siano utilizzate correttamente.

Nello Standard C l'operatore di dereferenza "`*`" nella chiamata a funzione tramite puntatore a funzione può essere omesso, di conseguenza la chiamata a funzione

```
tmp = (*fun)((double) k);
```

può essere scritta semplicemente come

```
tmp = fun((double) k);
```

Il cast esplicito del valore della variabile `k` non è necessario e può essere omesso.

Nel linguaggio C l'identificatore di una funzione quando non è utilizzato per una chiamata a funzione, o non è argomento dell'operatore di indirizzo "`&`" o dell'operatore `sizeof`, è convertito automaticamente nel tipo "puntatore a funzione". Di conseguenza un parametro formale dichiarato nel prototipo di funzione di tipo "funzione che ritorna il tipo `T`" viene automaticamente convertito in "puntatore a funzione che ritorna il tipo `T`" cosicché, ad esempio, nella definizione della funzione `sum_func_square()` il prototipo può anche essere scritto come:

```
double sum_func_square(double fun(double), int n)
```

Di seguito sono riportati alcune dichiarazioni equivalenti della funzione `sum_func_square()`:

```
double sum_func_square(double fun(double x), int n);
double sum_func_square(double fun(double), int n);
double sum_func_square(double fun(double x), int);
double sum_func_square(double fun(double), int);
double sum_func_square(double (*fun)(double), int);
double sum_func_square(double (*)(double), int);
```

Nei primi quattro prototipi l'identificatore `fun` non può essere eliminato perché fa parte della dichiarazione del tipo funzione, il prototipo seguente

```
double sum_func_square(double (double), int n);
```

è infatti illegale.

Il seguente programma illustra l'uso della funzione `sum_func_square()` sia con una funzione di sistema che con una fornita nel programma.

Programma: `test-sum_func_square.c`

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

```
static double square(double);
extern double sum_func_square(double (*fun)(double), int);

int main(void)
{

  printf("1) Somma di sin    : %.3f\n",
         sum_func_square(sin, 3));
  printf("2) Somma di square : %.3f\n",
         sum_func_square(square, 3));
  return 0;
}

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

Quando il programma viene compilato

```
$ cc test-sum_func_square.c sum_func_square.c -lm
```

ed eseguito si ha

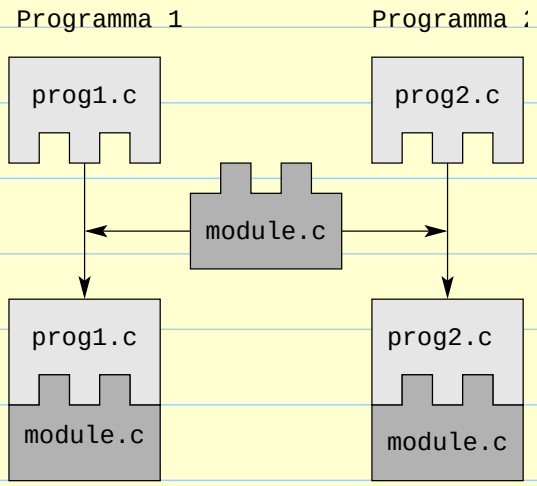
```
1) Somma di sin    : 1.555
2) Somma di square : 98.000
```

Incontreremo altri esempi di funzioni i cui parametri sono puntatori a funzioni in quanto la possibilità di poter utilizzare le funzioni come parametri di funzioni permette una programmazione molto flessibile.

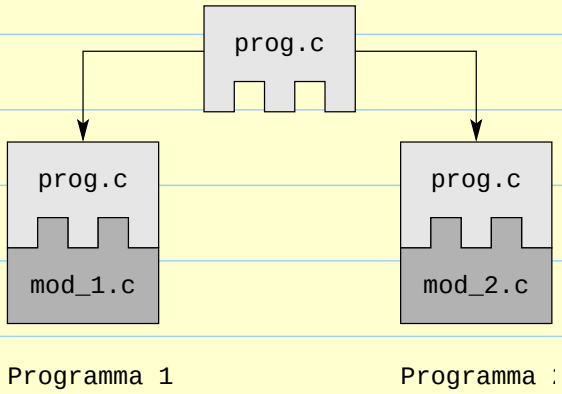
2.30. Programmazione modulare (Rev. 2.1)

La programmazione *modulare* è la naturale estensione della programmazione strutturata in cui una o più funzioni vengono raggruppate in un unico file sorgente a formare una unità indipendente. Nella programmazione modulare ciascun programma viene quindi suddiviso in più files sorgente ciascuno dei quali rappresenta un *modulo* del programma. Il programma finale è poi ottenuto "assemblando" insieme i vari moduli.

Il grosso vantaggio della programmazione modulare è che è possibile sviluppare moduli utilizzabili per costruire più programmi diversi



riducendo notevolmente i tempi di sviluppo dei programmi. Un esempio è la libreria matematica standard che riunisce funzioni matematiche come `sin()`, `cos()`, `pow()` e così via, o la libreria standard di Input/Output. La possibilità di strutturare un programma in moduli permette inoltre una grande flessibilità perché a volte per modificare un programma, od adattarlo ad un particolare sistema, è sufficiente la sostituzione di pochi moduli.



In linea di principio un modulo può essere formato da un insieme qualsiasi di funzioni che formano un programma, tuttavia la logica della programmazione modulare richiede che un modulo sia formato da un insieme di funzioni correlate o con caratteristiche comuni, ad esempio le funzioni matematiche nel caso della libreria matematica standard o le funzioni di Input/Output per la libreria di Input/Output standard. Inoltre è bene tener presente che non tutti i compilatori riescono ad effettuare operazioni di ottimizzazione tra file sorgente diversi. Di conseguenza sebbene non vi siano regole specifiche per dividere un programma in moduli né per scrivere moduli suggeriamo comunque di attenersi nella loro scrittura alle seguenti indicazioni di carattere generale:

- Tutte le funzioni di uno stesso modulo devono effettuare operazioni relative allo scopo del modulo.
- Le informazioni passate tra i moduli devono essere limitate.

Un modulo è composto da due files: il file sorgente, il cui nome termina con il suffisso "`.c`", ed il file di *header* che per convenzione ha lo stesso nome del file sorgente del modulo ma termina con il suffisso "`.h`".

Il file sorgente contiene le istruzioni e quanto serve per definire le funzioni che formano il modulo ed è composto da una parte *pubblica* ed una parte *privata*. La parte pubblica comprende tutte le funzioni, le variabili e quant'altro deve essere visibile al di fuori del file sorgente del modulo e quindi utilizzabile in altri files sorgente. La parte privata comprende invece tutto ciò che non deve essere visibile fuori del file sorgente del modulo. Di conseguenza le dichiarazioni delle variabili private a scopo globale e le definizioni delle funzioni private devono essere fatte con la classe di memorizzazione `static`.

Il file di header contiene tutte le informazioni sulla parte pubblica del modulo necessarie per utilizzare le sue funzioni. In particolare il file di header deve contenere le costanti e le definizioni pubbliche, i prototipi delle funzioni pubbliche e le dichiarazioni delle variabili pubbliche. È buona norma inoltre includere nel file di header anche informazioni sul contenuto ed utilizzo del modulo.

Il file di header deve essere incluso in ogni files sorgente che utilizzi il modulo con il comando del preprocessore "`#include`" seguito dal nome del file di header tra due coppie di doppi apici "*header_file*`.h`", non essendo il file di header un file di header di sistema. Ad esempio per utilizzare le funzioni matematiche è necessario includere il file di header `math.h` mentre per quelle di Input/Output il file di header `stdio.h`. In questo caso si usano i delimitatori "`<...>`" perché si tratta di files di sistema.

Siccome le variabili e le funzioni dichiarate nel file di header sono definite in un file diverso dal file sorgente che lo include, sono infatti definite nel file sorgente del modulo, tutte le dichiarazioni sia di variabili che di funzioni contenute nel file di header devono essere effettuate con la classe di memorizzazione `extern`.

L'operazione di "attaccare" insieme i vari moduli che formano un programma per produrre un eseguibile viene effettuata dal *linker* invocato dal compilatore C subito dopo la compilazione dei files sorgente che compongono il programma. Supponiamo ad esempio che il programma sia composto da un file principale `prog.c` che contiene la funzione `main()` e da un modulo `module.c`. Il programma eseguibile viene generato con le seguenti istruzioni

```
$ cc prog.c module.c -o prog
```

ovvero compilando separatamente i due files ed invocando poi il linker per "attaccarli" insieme e produrre l'eseguibile

```
$ cc -c prog.c
$ cc -c module.c
$ cc prog.o module.o -o prog
```

Ricordiamo che quando viene specificato il *flag* "`-c`" il processo di compilazione si arresta subito dopo la fase di compilazione e viene prodotto un file object con lo stesso nome del file sorgente ma con il suffisso "`.o`" (*object*).

Come esempio consideriamo il seguente semplice programma composto dal file sorgente principale

File: `prog.c`

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

int main(void)
{
  int ind;

  for (ind = 0; ind < 20; ++ind) {
    inc_count();
  }
  printf("Il contatore vale %d\n", Count);

  return 0;
}
```

ed dal modulo

File: `module.c`

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

/* Contatore cicli: globale esterna */
int Count = 0;

/* funzione privata modulo */
static void do_something(int c);

void inc_count(void)
{
  ++Count;
  if (Count % 5 == 0) do_something(Count);
  return;
}

static void do_something(int c)
{
  printf("Hey boy, count reached %3d!\n", c);
  return;
}
```

In questo caso il modulo contiene la sola funzione pubblica `inc_count()` che incrementa la variabile pubblica `Count` ogni volta che la funzione viene chiamata, e da una funzione privata `do_something()` che nell'esempio semplicemente scrive un messaggio sullo `stdout`. Per indicare che la variabile `Count` ha scopo globale il suo identificatore è scritto con il primo carattere maiuscolo. Sebbene questo non sia obbligatorio è una buona norma di programmazione quella di utilizzare una convenzione per gli identificatori delle variabili a scopo globale in modo da distinguerle facilmente.

Osserviamo che lo specificatore `static` nella definizione della funzione `do_something()` si sarebbe potuto omettere perché la definizione è preceduta dalla dichiarazione della funzione in classe di memorizzazione `static`.

Il file di header del modulo è

File: `module.h`

```
/* numero di cicli effettuati */
extern int Count;

/* contatore cicli */
extern void inc_count(void);
```

Questo viene incluso nel file sorgente `prog.c` con il comando del preprocessore

```
#include "module.h"
```

poiché il file di header si trova nello stesso direttorio del file sorgente `prog.c`.

Il file di header viene incluso anche nel file sorgente `module.c` per permettere al compilatore di controllare la consistenza tra il prototipo della dichiarazione della funzione `inc_count()` contenuta nel file di header `module.h` con il prototipo della definizione della funzione contenuta nel file `module.c`. Non vi è conflitto tra la dichiarazione

```
extern int Count;
```

contenuta nel file di header `module.h` e la dichiarazione

```
int Count = 0;
```

contenuta nel file `module.c` poiché la prima viene interpretata come una referenza alla variabile `Count` mentre la seconda come la definizione della variabile `Count`.

Il programma viene compilato con l'istruzione

```
$ cc prog.c module.c -o prog
```

e quando viene eseguito produce il seguente output

```
Hey boy, count reached   5!
Hey boy, count reached  10!
Hey boy, count reached  15!
Hey boy, count reached  20!
Il contatore vale 20
```

È buona norma utilizzare sempre la classe `static` per le variabili globali private altrimenti si può incorrere in errori di difficile individuazione come mostra il seguente esempio composto dal file sorgente principale

File: `prog_bad.c`

```
#include <stdio.h>

int var = 0;
```

```
int main(void)
{
  printf("var vale: %d\n", var);
  return 0;
}
```

e dal modulo

File: `module_bad.c`

```
int var = 1;
```

La variabile `var` è visibile in entrambi i files, cosa succede in questo caso? La risposta dipende dal compilatore. In alcuni casi fortunati il compilatore segnala che la variabile è definita più di una volta mentre in altri, meno fortunati, il risultato può essere sia `1` che `0` a seconda di quale delle due definizione è stata fatta per prima.

Suggerimenti per la scrittura dei moduli

La scrittura di un programma su più files sorgente limita i controlli che possono essere effettuati dal compilatore. Ad esempio il compilatore C non può controllare che i prototipi delle dichiarazioni di una stessa funzione in files differenti siano compatibili tra loro e con la definizione della funzione, o che gli argomenti della funzione in una chiamata siano del tipo richiesto dalla definizione della funzione. Se al posto dei files `prog.c` e `module.c` fossero stati usati i files

File: `prog_wrong.c`

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

int do_something(void);

int main(void)
{
  int ind;

  for (ind = 0; ind < 20; ++ind) {
    inc_count();
  }
  printf("Il contatore vale %d\n", Count);
  printf("do_something -> %d\n", do_something());

  return 0;
}
```

e

File: `module_wrong.c`

```c
#include <stdio.h>

/* Contatore cicli: globale esterna */
int Count = 0;

/* funzione privata modulo */
void do_something(int c);

void inc_count(void)
{
  ++Count;
  if (Count % 5 == 0) do_something(Count);
  return;
}

void do_something(int c)
{
  printf("Hey boy, count reached %3d!\n", c);
  return;
}
```

il compilatore non avrebbe segnalato nessun problema.

In questi casi il risultato è imprevedibile. Nei casi più fortunati il programma termina inaspettatamente o produce risultati senza senso, in quelli meno fortunati il programma semplicemente produce risultati plausibili ma errati.

Errori di questo tipo possono essere molto difficili da trovare, tuttavia è possibile prevenirli attenendosi alle seguenti semplici regole.

- Tutte le funzioni esterne devono avere una sola dichiarazione in forma di prototipo in un file di header. La dichiarazione dovrebbe contenere esplicitamente la classe di memorizzazione `extern`. In questo modo viene elimiata la possibilità di dichiarazioni con prototipi incompatibili.
- In ogni file sorgente che utilizza una funzione esterna deve essere incluso il file di header con la dichiarazione della funzione. In questo modo le chiamate alla funzione sono controllate tutte dallo stesso prototipo della funzione cosicchè il compilatore può controllare la consistenza degli gli argomenti nelle differenti chiamate con i parametri formali del prototipo della funzione.
- Il file sorgente che contiene la definizione di una funzione esterna dovrebbe includere il file di header con la dichiarazione della funzione in modo che il compilatore possa controllare la consistenza tra il prototipo della definizione con quello della dichiarazione contenuto nel file di header. Questo assicura, insieme alla regola precedente, che tutte le chiamate della funzione siano consistenti con la definzione.
- Prima della definizione e di qualsiasi chiamata di una funzione statica vi dovrebbe essere una dichiarazione in forma di prototipo della funzione. Ricordiamo che le funzioni statiche devono essere dichiarate esplicitamente con calsse di memorizzazione

`static`. Questo assicura che l'identificatore della funzione sia associato con la corretta definizione.

Se ci si attiene a queste regole difficilmente si avranno problemi causati da incompatibilità tra le chiamate e le definizioni delle funzioni.

2.31. Esempio: Modulo di integrazione numerica (Rev. 2.1.1)

Come esempio di programmazione modulare consideriamo la scrittura di un modulo di integrazione numerica per integrali della forma

$$I = \int_a^b f(x)\,dx$$

Il modulo fornisce le funzioni:

```
#include "integ.h"

double integ_rett(double a, double b, int n_int,
                  double (*f)(double));

double integ_trap(double a, double b, int n_int,
                  double (*f)(double));

double integ_simp(double a, double b, int n_int,
                  double (*f)(double));

double integ_mc(double a, double b, int n_mcs, int seed,
                double (*f)(double));
```

Descrizione: La funzione `integ_rett()` restituisce il valore dell'integrale della funzione `f()` tra gli estremi `a` e `b` valutato con il metodo dei rettangoli dividendo l'intervallo `[a,b]` in `n_int` intervalli. Se il valore di `n_int` è 1 l'intervallo di integrazione non viene suddiviso.

La funzione `integ_trap()` restituisce il valore dell'integrale della funzione `f()` tra gli estremi `a` e `b` valutato con il metodo dei trapezi dividendo l'intervallo `[a,b]` in `n_int` intervalli. Se il valore di `n_int` è 1 l'intervallo di integrazione non viene suddiviso.

La funzione `integ_simp()` restituisce il valore dell'integrale della funzione `f()` tra gli estremi `a` e `b` valutato con il metodo di Simpson dividendo l'intervallo `[a,b]` in `n_int` intervalli. Il valore di `n_int` deve essere un intero positivo pari.

La funzione `integ_mc()` restituisce il valore dell'integrale della funzione `f()` tra gli estremi `a` e `b` valutato con il metodo Monte Carlo con `n_mcs` punti. Il parametro `seed` inizializza il generatore di numeri aleatori (*random*) usato dalla funzione. Se il valore di `seed` è 0 il generatore non viene inizializzato.

Valore restituito: Tutte le funzioni restituiscono il valore dell'integrale o `NaN` (*not-a-number*) se il valore di `n_int` o `n_mcs` è negativo o nullo o, nel caso della funzione `integ_simp()`, dispari.

Files: Il modulo è composto dal un file sorgente `module.c` e dal file di header `integ.h`.

File sorgente: `integ.c`

```
/*
 * Descrizione :  Modulo di integrazione numerica
 *
 * Funzioni: double integ_rett(double a, double b, int n_int,
 *                             double (*f)(double));
 *           double integ_trap(double a, double b, int n_int,
 *                             double (*f)(double));
 *           double integ_simp(double a, double b, int n_int,
 *                             double (*f)(double));
 *           double integ_mc(double a, double b, int n_mcs, int seed,
 *                           double (*f)(double));
 *
 * $yaC-Primer: integ.c, v 2.2 02.03.05 AC $
 */
#include <stdlib.h>
#include <math.h>

#ifndef INTEG_H
#include "integ.h"
#endif

/* NaN privato */
static double my_nan(void);

/* ----
 * integ_rett()
 *
 * Integrazione metodo dei rettangoli
 */
extern double integ_rett(double a, double b, int n_int,
                         double (*f)(double))
{
  int         it;      /* indice intervallo                */
  double sum, dx;      /* integrale ed ampiezza intervalli */
  double   x_tmp;

  /* consistenza numero intervalli */
  if (n_int <= 0) return my_nan();

  /* ampiezza intervalli  */
  dx = (b - a) / (double) n_int;

  /* integrazione */
  sum = 0.0;
  x_tmp = a + 0.5 * dx;
  for (it = 1; it <= n_int; ++it) {
    sum   += (*f)(x_tmp);
    x_tmp += dx;
```

```
  }

  return (sum * dx);
}

/* ----
 * integ_trap()
 *
 * Integrazione metodo dei trapezi
 */
extern double integ_trap(double a, double b, int n_int,
                         double (*f)(double))
{
  int         it;      /* indice intervallo                */
  double sum, dx;      /* integrale ed ampiezza intervalli */
  double   x_tmp;

  /* consistenza numero intervalli */
  if (n_int <= 0) return my_nan();

  /* ampiezza intervalli  */
  dx = (b - a) / (double) n_int;

  /* integrale */
  sum = 0.5 * ( f(a) + f(b) );
  x_tmp = a;
  for (it = 1; it < n_int; ++it) {
    x_tmp  += dx;
    sum    += f(x_tmp);
  }

  return (sum * dx);
}

/* ----
 * integ_simp()
 *
 * Integrazione metodo di Simpson
 */
extern double integ_simp(double a, double b, int n_int,
                         double (*f)(double))
{
  int           it, n;   /* indice intervalli e # intervalli */
  double      sum, dx;   /* integrale ed ampiezza intervalli */
  double sum_e, sum_o;   /* somma termini pari e dispari     */
  double        x_tmp;

  /* consistenza numero intervalli */
  if ( (n_int <= 0) || (n_int % 2) ) return my_nan();

  /* # intervalli amp. 2 dx */
  n  = n_int / 2;
```

```
  dx = (b - a) / (double) n_int;

  /* integrale: separatamente pari e dispari */
  x_tmp = a + dx;
  sum_e = (double) 0.0;
  sum_o = f(x_tmp);
  for (it = 1; it < n; ++it) {
    x_tmp  += dx;
    sum_e  += f(x_tmp);     /* indice pari    -> 2 */
    x_tmp  += dx;
    sum_o  += f(x_tmp);     /* indice dispari -> 4 */
  }

  sum = (f(a) + 2.0 * sum_e + 4.0 * sum_o + f(b)) / 3.0;
  return (sum * dx);
}

/* ----
 * integ_mc()
 *
 * Integrazione metodo Monte Carlo
 */
extern double integ_mc(double a, double b, int n_mcs, int seed,
                       double (*f)(double))
{
  int         mc;   /* indice sample                          */
  double sum, dx;   /* integrale ed ampiezza intervallo int. */
  double   x_tmp;

  /* consistenza numero samples  */
  if (n_mcs <= 0) return my_nan();

  /* inizializzazione generatore di numeri random */
  if (seed) srand(seed);

  /* intervallo integrazione */
  dx = b - a ;

  /* integrale */
  sum = 0.0;
  for (mc = 0; mc < n_mcs; ++mc) {
    x_tmp = a + dx * (double) rand() / (RAND_MAX + 1.0);
    sum   += (*f)(x_tmp);
  }

  return (dx * sum / (double) n_mcs);
}

/* -------------------------- */
/* Funzioni private del modulo */
/* -------------------------- */
static double my_nan(void)
```

```
{
  return strtod("NAN", NULL);
}
```

Note sul file: `integ.c`

- `static double my_nan(void);`

  La funzione `my_nan()` restituisce il valore `NaN` (*not-a-number*) ottenuto utilizzando la chiamata a funzione

  ```
  strtod("NAN", NULL);
  ```

  Si è già incontrato il valore `NaN` e la funzione `strtod()` che converte una stringa in valore numerico di tipo `double` discutendo il metodo della bisezione, si rimanda per cui a quella sezione per maggiori informazioni.

  La funzione `my_nan()` è stata introdotta nel modulo al solo scopo di illustrare l'uso delle funzioni private poiché è chiaramente possibile sostituire ogni istruzione

  ```
  return my_nan();
  ```

  direttamente con

  ```
  return strtod("NAN", NULL);
  ```

Funzione `integ_rett()`

- ```
  sum = 0.0;
  x_tmp = a + 0.5 * dx;
  for (it = 1; it <= n_int; ++it) {
    sum   += (*f)(x_tmp);
    x_tmp += dx;
  }
  ```

  Il punto medio del primo intervallo è ottenuto aggiungendo all'estremo di integrazione `a` metà della lunghezza `dx` degli intervalli in cui è stato suddiviso l'intervallo di integrazione $[a, b]$. Il punto medio di ciascun intervallo successivo è ottenuto semplicemente aggiungendo la lunghezza dell'intervallo `dx` al punto medio dell'intervallo precedente. L'ultimo valore di `x_tmp` giace oltre `b` ma non essendo utilizzato non crea nessun problema nel calcolo dell'integrale.

Funzione `integ_trap()`

- ```
  sum = 0.5 * ( f(a) + f(b) );
  x_tmp = a;
  for (it = 1; it < n_int; ++it) {
    x_tmp  += dx;
    sum    += f(x_tmp);
  }
  ```

Gli estremi `a` e `b` dell'intervallo di integrazione sono considerati separatamente ed il ciclo `for` va dal secondo intervallo di lunghezza `dx` in cui è stato suddiviso l'intervallo di integrazione $[a, b]$ fino al penultimo. Il ciclo viene infatti eseguito `n_int - 1` volte.

Funzione `integ_simp()`

- `if ( (n_int <= 0) || (n_int % 2) ) return my_nan();`

Il metodo di Simpson richiede che l'intervallo di integrazione $[a, b]$ sia diviso in un numero pari di intervalli. Se il valore di `n_int` non è pari la funzione restituisce il valore `NaN`, tuttavia altre strategie sono possibili.

- ```
  x_tmp = a + dx;
  sum_e = (double) 0.0;
  sum_o = f(x_tmp);
  for (it = 1; it < n; ++it) {
    x_tmp  += dx;
    sum_e  += f(x_tmp);      /* indice pari    -> 2 */
    x_tmp  += dx;
    sum_o  += f(x_tmp);      /* indice dispari -> 4 */
  }
  ```

Per limitare il numero di moltiplicazioni si sommano separatamente il valore della funzione sui punti di indice pari (`sum_e`) e quelli di indice dispari (`sum_o`). Il primo intervallo di estremi $x_0 = a$ e $x_1 = a + dx$ è considerato separatamente dagli altri. Il valore dell'integrale secondo la formula di Simpson è dato da

```
sum = (f(a) + 2.0 * sum_e + 4.0 * sum_o + f(b)) / 3.0;
```

moltiplicato per la lunghezza `dx` di ciascun intervallo in cui è stato suddiviso l'intervallo di integrazione $[a, b]$.

Funzione `integ_monte()`

- `if (seed) srand(seed);`

Il metodo Monte Carlo utilizza un *generatore di numeri*
slcrandom per scegliere i punti su cui valutare la funzione da integrare. Tutti i generatori di numeri aleatori vanno inizializzati prima di poterli usare. La funzione

```
#include <stdlib.h>

void srand(unsigned seed);
```

inizializza il generatore di numeri `rand()` utilizzando il valore del *seme* `seed`. Se il valore di `seed` è uguale a `0` il generatore non viene inizializzato per permettere di riutilizzare la funzione `integ_monte()` con sequenze di numeri differenti.
```

- ```
  sum = 0.0;
  for (mc = 0; mc < n_mcs; ++mc) {
    x_tmp = a + dx * (double) rand() / (RAND_MAX + 1.0);
    sum   += (*f)(x_tmp);
  }
  ```

  La funzione `integ_mc()` sceglie i punti su cui calcolare la funzione uniformemente nell'intervallo $[a, b]$. A questo scopo utilizza la funzione

  ```
  #include <stdlib.h>

  int rand(void);
  ```

  che restituisce un numero di tipo `int` distribuito uniformemente tra `0` e `RAND_MAX` inclusi cosicché

  ```
  x_tmp = a + dx * (double) rand() / (RAND_MAX + 1.0);
  ```

  è distribuito uniformemente tra `a` e `b` (escluso). La macro `RAND_MAX` è definita nel file di header `stdlib.h` con un valore che dipende dal sistema. Osserviamo che essendo sia `RAND_MAX` che il valore restituito dalla funzione `rand()` di tipo `int` la divisione deve essere effettuata esplicitamente in floating-point altrimenti il risultato è sempre zero.

File di header: `integ.h`

```
/*
 * Header file modulo integ.c
 *
 * Funzioni: double integ_rett(double a, double b, int n_int,
 *                             double (*f)(double))
 *           double integ_trap(double a, double b, int n_int,
 *                             double (*f)(double))
 *           double integ_simp(double a, double b, int n_int,
 *                             double (*f)(double))
 *           double integ_mc(double a, double b, int n_mcs, int seed,
 *                           double (*f)(double))
 *
 * Descrizione :
 *
 *  integ_rett() -> Integrazione rettangoli
 *  integ_trap() -> Integrazione trapezi
 *  integ_simp() -> Integrazione Simpson
 *  integ_mc()   -> Integrazione Monte Carlo
 *
 *  a, b  -> estremi intervallo integrazione
 *  n_int -> numero divisioni intervallo
 *  n_mcs -> numero Montecarlo samples
 *  seed  -> seme generatore numeri random
 *  f     -> funzione da integrare
 *
 * $yaC-Primer: integ.h, v 2.2 23.02.05 AC $
 */
```
```

```
#ifndef INTEG_H
#define INTEG_H
#endif

extern double integ_rett(double a, double b, int n_int,
                         double (*f)(double));

extern double integ_trap(double a, double b, int n_int,
                         double (*f)(double));

extern double integ_simp(double a, double b, int n_int,
                         double (*f)(double));

extern double integ_mc(double a, double b, int n_mcs, int seed,
                       double (*f)(double));
```

Il file di header, oltre alle dichiarazioni delle funzioni ed una descrizione del modulo, contiene la definizione della macro `INTEG_H` che può essere utilizzata per evitare inclusioni multiple del file `integ.h` includendolo ogni qual volta serve con le istruzioni

```
#ifndef INTEG_H
#include "integ.h"
#endif
```

Test del modulo `integ.c`: Il seguente programma mostra l'uso del modulo con la funzione di prova $f(x) = x^2$.

Programma: `test-integ.c`

```
/*
 * Descrizione :  test modulo integ.c
 *
 * $yaC-Primer: test-integ.c, v 2.1 02.03.2005 AC $
 */
#include <stdio.h>
#include <math.h>
#include <stdio.h>
#ifndef INTEG_H
#include "integ.h"
#endif

/* funzione di prova */
static double f(double);

int main(void)
{
  int     n_int;
  int     n_sam;
  double  value;
```

```
  n_int = 10;

  value = integ_rett(0, 1.0, n_int, f);
  printf("\n");
  printf ("Rettangoli con %d intervalli : %f\n\n",
          n_int, value);

  value = integ_trap(0, 1.0, n_int, f);
  printf ("Trapezi con %d intervalli    : %f\n\n",
          n_int, value);

  value = integ_simp(0, 1.0, n_int, f);
  printf ("Simpson con %d intervalli    : %f\n\n",
          n_int, value);

  n_sam = 10000;
  value = integ_mc(0, 1.0, n_sam, 12345, f);
  printf ("Monte Carlo con %d punti     : %f\n\n",
          n_sam, value);

  value = integ_mc(0, 1.0, n_sam, 0, f);
  printf ("Monte Carlo con %d punti     : %f\n\n",
          n_sam, value);

  return 0;
}

static double f(double x)
{
  return (x*x);
}
```

Il programma viene compilato con le istruzioni

```
$ cc test-integ.c integ.c -lm
```

e quando viene eseguito produce il seguente output:

```
Rettangoli con 10 intervalli : 0.332500

Trapezi con 10 intervalli    : 0.335000

Simpson con 10 intervalli    : 0.333333

Monte Carlo con 10000 punti      : 0.328321

Monte Carlo con 10000 punti      : 0.331105
```

Osserviamo che il valore ottenuto con il metodo di Simpson è quello corretto $1/3$. Il metodo è infatti esatto per polinomi fino al terzo ordine incluso. Se come funzione di test si utilizzasse $f(x) = x$ anche i risultati ottenuti con il metodo dei rettangoli e dei trapezi sarebbero esatti.

Il metodo Monte Carlo è un metodo statistico di conseguenza con un numero finito di punti il risultato dipende dai punti scelti e quindi le due chiamate successive della funzione `integ_mc()` producono risultati differenti ma consistenti entro l'errore statistico che con 10^4 punti è di ordine $O(10^{-2})$.

2.32. Funzione `main()` (Rev. 2.1)

Ogni programma scritto il linguaggio C, anche se composto da più files sorgente o moduli, deve contenere una ed una sola definizione della funzione `main()`. La funzione `main()` non è una funzione di libreria ma la funzione "principale" da cui inizia l'esecuzione del programma, in altre parole è la funzione che viene eseguita per prima. Quando termina l'esecuzione della funzione `main()` termina anche l'esecuzione del programma ed il valore restituito dalla funzione viene utilizzato come un *codice di errore* per indicare se il programma è terminato con successo o no. Se l'esecuzione del programma raggiunge la fine del corpo della funzione `main()` senza che sia stata incontrata un'istruzione `return` viene assunto per *default* che l'esecuzione della funzione sia stata terminata dall'istruzione `return 0;` subito prima della fine del corpo.

Nello Standard C89 se il tipo della funzione `main()` non viene specificato viene utilizzato per *default* il tipo `int`. Nello Standard C99 questo non è permesso e il tipo `int` deve essere specificato esplicitamente. La funzione `main()` è sempre di tipo `int`.

Nello Standard C la funzione `main()` può essere definita sia senza parametri:

```
int main(void) { ... }

int main() { ... }          /* Non raccomandato */

main() { ... }              /* Non raccomandato. C99 Non permesso */
```

o con due parametri chiamati usualmente `argc` e `argv`,

```
int main(int argc, char *argv[]) { ... }
```

il cui valore viene assegnato dal sistema operativo quando inizia l'esecuzione del programma.

Il primo parametro `argc` di tipo `int` è il contatore del numero di "argomenti" o "opzioni" passati al programma quando questo viene invocato sia dalla linea di comando che da un altro programma. Questi argomenti sono chiamati argomenti di linea (*command line arguments*) perché sono dati di seguito al nome dell'eseguibile del programma sulla stessa linea di comando.

Il secondo parametro `argv` è un array di puntatori a stringhe contenenti gli argomenti di linea passati al programma. La prima stringa `argv[0]` contiene il nome del programma o, se questo non è disponibile, la stringa vuota. In questo caso `argv[0][0]` contiene il carattere '`\0`'. Le altre `argc-1` stringhe contengono gli argomenti di linea passati al programma ordinati con l'*i*-esimo argomento nell'*i*-esima stringa `argv[i]`. Lo Standard C richiede inoltre che `argv[argc]` sia il puntatore nullo `NULL`.

Il seguente programma mostra il contenuto delle stringhe `argv[i]` fino a quello di `argv[argc]`

incluso.

Programma: `comm_line.c`

```
#include <stdio.h>

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

  printf("\n argc = %d\n\n", argc);

  printf(" Nome         -> argv[%d] = \"%s\"\n", 0, argv[0]);
  for (i = 1; i < argc; ++i) {
    printf(" Argomenti    -> argv[%d] = \"%s\"\n", i, argv[i]);
  }

  printf(" Terminatore -> argv[%d] = %s\n", i, argv[i]);
  printf("\n");

  return 0;
}
```

Quando il programma viene eseguito si ha

```
$ a.out argomenti di linea

 argc = 4

 Nome        -> argv[0] = "a.out"
 Argomenti   -> argv[1] = "argomenti"
 Argomenti   -> argv[2] = "di"
 Argomenti   -> argv[3] = "linea"
 Terminatore -> argv[4] = (null)
```

Il linguaggio C converte automaticamente i parametri formali dei prototipi delle funzioni dichiarati di tipo "array di tipo `T`" in "puntatore a tipo `T`", di conseguenza nella definizione della funzione `main()` si può utilizzare anche il prototipo equivalente

```
int main(int argc, char **argv) { ... }
```

Ricordando che una stringa è un array di caratteri è facile convincersi che `argv[i][j]` rappresenta il j-esimo carattere (o elemento) dell'i-esima stringa cosicchè `argv` può essere pensato come una array bidimensionale in cui `argv[i]` punta alla i-esima riga del'array, come mostrato dal seguente programma che stampa i comandi di linea del programma prima come stringa e poi carattere per carattere. trattando `argv` come se fosse un array bidimensionale.

Programma: `array_pointers.c`

```
#include <stdio.h>

int main(int argc, char **argv)
```

```
{

  int i, j;

  printf("\n argc = %d\n\n", argc);

  for (i = 0; i < argc; ++i) {
    printf(" argv[%d] = \"%s\"\n", i, argv[i]);
    for ( j = 0; argv[i][j] != '\0'; ++j) {
      printf(" --> argv[%d][%d] = '%c'\n", i, j, argv[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

Quando il programma viene eseguito produce un output del tipo:

```
$ ./a.out line arguments

  argc = 3

  argv[0] = "./a.out"
  --> argv[0][0] = '.'
  --> argv[0][1] = '/'
  --> argv[0][2] = 'a'
  --> argv[0][3] = '.'
  --> argv[0][4] = 'o'
  --> argv[0][5] = 'u'
  --> argv[0][6] = 't'

  argv[1] = "line"
  --> argv[1][0] = 'l'
  --> argv[1][1] = 'i'
  --> argv[1][2] = 'n'
  --> argv[1][3] = 'e'

  argv[2] = "arguments"
  --> argv[2][0] = 'a'
  --> argv[2][1] = 'r'
  --> argv[2][2] = 'g'
  --> argv[2][3] = 'u'
  --> argv[2][4] = 'm'
  --> argv[2][5] = 'e'
  --> argv[2][6] = 'n'
  --> argv[2][7] = 't'
  --> argv[2][8] = 's'
```

Nonostante questa somiglianza `argv` è un oggetto diverso da un array bidimensionale, il primo infatti è un "array di puntatori" mentre il secondo un "array di arrays". Una conseguenza di questa differenza è che ad esempio la lunghezza delle righe di `argv` può variare da riga a riga

mentre in un array bidimensionale queste sono tutte della stessa lunghezza. Un array le cui righe possono essere di lunghezza differente viene anche chiamato *array frastagliato*.

Il sistema operativo UNIX utilizza i caratteri '`*`' e '`?`' come *widlcards* nei nomi dei files per indicare rispettivamente "ogni sequenza di caratteri" ed "ogni singolo carattere". Se questi caratteri compaiono negli argomenti di linea il loro valore viene sostituito dal sistema operativo prima di passarlo al programma come argomento di linea, per cui nel caso dei due precedenti programmi il comando

```
$ ./a.out *
```

stamperebbe la lista di tutti i files presenti nel direttorio, il comando

```
$ ./a.out ???
```

il nome di tutti i files il cui nome è composto da tre caratteri, mentre il comando

```
$ ./a.out a*
```

il nome di tutti i files il cui nome è inizia con il carattere '`a`' e così via.

Alcuni sistemi permettono di definire la funzione `main()` con un terzo argomento chiamato usualmente `char *env[]` o `char *envp[]` che punta ad un array di stringhe che contengono i valori dei parametri di configurazione (*environment parameters*) dell'utente come ad esempio il direttorio in cui ci si trova o la username. Alcuni parametri tipici sono

`USER`	login dell'utente
`LANG`	nome della lingua usata
`LOGNAME`	login dell'utente
`HOME`	direttorio principale dell'utente
`HOST`	nome del computer
`PATH`	sequenza dei direttori in cui cercare un file se non viene specificata la sua locazione
`PWD`	direttorio in cui ci si trova

I parametri sono scritti nella nella forma "`param_name=value`".

L'array puntata da `env` è terminata dal puntatore nullo `NULL`, ossia contiene come ultimo elemento il puntatore nullo `NULL`.

Il seguente programma stampa il valore dei parametri di configurazione dell'utente.

Programma: `env_param.c`

```
#include <stdio.h>

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

  for (i = 0; env[i] != NULL; ++i) {
    printf(" env[%d] = \"%s\"\n", i, env[i]);
  }
```

```
  return 0;
}
```

Su alcuni sistemi UNIX è possibile accedere ai parametri di configurazione anche attraverso la variabile globale `char **environ`, o `char *environ[]` a seconda del sistema, come mostrato dal seguente programma.

Programma: `environ_param.c`

```
#include <stdio.h>

int main(void)
{
  int i;
  extern char **environ;

  for (i = 0; environ[i] != NULL; ++i) {
    printf(" environ[%d] = \"%s\"\n", i, environ[i]);
  }
  return 0;
}
```

L'uso di `env` o di `environ` non è tuttavia Standard C per cui per evitare problemi di portabilità conviene usare la funzione Standard C

```
#include <stdlib.h>

char *getenv(const char *name);
```

che prende come parametro il puntatore alla stringa `name` che contiene il nome del parametro di configurazione e restituisce il puntatore ad un'altra stringa che contiene il "valore" del parametro, o il puntatore `NULL` se il parametro non esiste. Il seguente programma che prende come argomenti di linea i nomi dei parametri di configurazione e ne stampa il valore mostra l'uso della funzione `getvenv()`

Programma: `getenv_param.c`

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

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

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

  return 0;
}
```

Ad esempio

```
$ ./a.out LANG LONG
 LANG = en_US
 LONG = (null)
```

mostra che il valore del parametro `LANG` è `en_US` mentre il parametro `LONG` non esiste. È bene ricordarsi che il nome ed il valore dei parametri di configurazione dipende dal sistema.

2.33. Arrays di puntatori e Puntatori a puntatori (Rev. 2.1)

Nel linguaggio C gli identificatori delle variabili di tipo "array di tipo `T`" sono convertiti automaticamente in "puntatori a tipo `T`" ed è per questo, ad esempio, che nella definizione della funzione `main()` si può di utilizzare sia il prototipo

```
int main(int argc, char *argv[]) { ... }
```

che il prototipo

```
int main(int argc, char **argv) { ... }
```

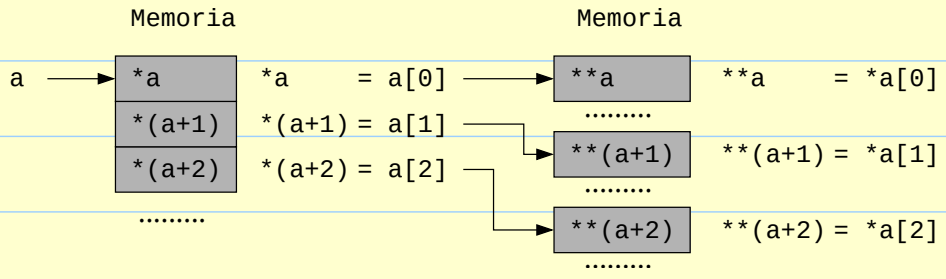
Da un punto di vista sintattico, tuttavia, le due dichiarazioni del parametro formale `argv` non sono equivalenti. Nel primo caso infatti `argv` viene dichiarato di tipo "array di puntatori a tipo `char`" mentre nel secondo caso viene dichiarato di tipo "puntatore a puntatore a tipo `char`".

Sebbene in questo contesto le due dichiarazioni siano perfettamente equivalenti una variabile di tipo "array di puntatori a tipo `T`" ed una di tipo "puntatore a puntatore a tipo `T`" sono oggetti diversi perché l'organizzazione in memoria degli oggetti a cui si riferiscono è differente.

Nel caso dell'array di puntatori a tipo `T`

```
T *a[size];
```

la struttura in memoria è:

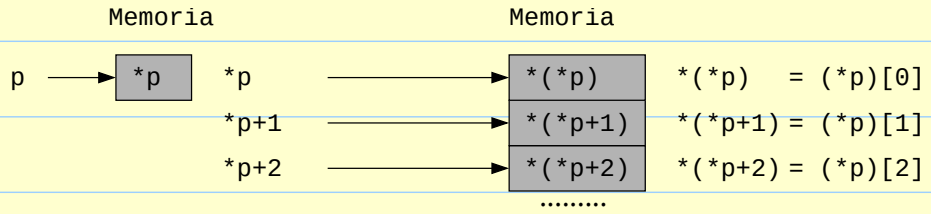


Infatti l'identificatore `a` dell'array è un puntatore al primo elemento di un array i cui elementi sono puntatori ad oggetti di tipo `T`, per cui il primo elemento dell'array `*a = a[0]` punta all'oggetto il cui valore è `**a = *a[0]`, il secondo elemento dell'array `*(a+1) = a[1]` punta all'oggetto il cui valore è `**(a+1) = *a[1]` e così via. Osserviamo che mentre i puntatori `a[0]`, `a[1]`, ..., `a[size-1]` occupano locazioni di memoria contigue gli oggetti puntati `*a[0]`, `*a[1]`, ..., `*a[size-1]` non necessariamente sono contigui in memoria.

Nel caso del puntatore a puntatore a tipo `T`

```
T **p;
```

la struttura in memoria è invece:



In questo caso infatti `p` è un puntatore ad una locazione di memoria che contiene il puntatore `*p` ad un oggetto di tipo `T` il cui valore è `*(*p) = (*p)[0]`, di conseguenza `*p+1` punta ad un oggetto di tipo `T` che si trova nella locazione di memoria subito successiva a quello puntato da `*p` ed il cui valore è `*(*p+1) = (*p)[1]`, `*p+2` punta ad un oggetto di tipo `T` che si trova nella locazione di memoria subito successiva a quello puntato da `*p+1`, ovvero due locazioni di memoria dopo quello puntato da `*p`, ed il cui valore è `*(*p+2) = (*p)[2]`, e così via. In altre parole `*p` è un puntatore al primo elemento di un array di tipo `T` e quindi in questo caso sono gli oggetti di tipo `T` che occupano locazioni di memoria contigue.

Il seguente programma esemplifica questa differenza

Programma: `arrayp_vs_ptrptr.c`

```c
#include <stdio.h>

int main(void)
{
  int        i;
  int       a1 = 1, a2 = 2, a3 = 3;
  int array[5] = {10, 20, 30, 40, 50};
  int       a4 = 4, a5 =  5;
  int    *a[5];               /* array di puntatori a int    */
  int       *q;               /* puntatore a int             */
  int      **p;               /* puntatore a puntatore a int */

  a[0] = &a1;                 /* puntatori a var a1, ..., a5 */
  a[1] = &a2;
  a[2] = &a3;
  a[3] = &a4;
  a[4] = &a5;

  q = array;                  /* q punta ad array */
  p = &q;                     /* p punta a q      */

  for (i = 0; i < 5; ++i)
    printf("*a[%1d]= %2d (%lu) \t (*p)[%1d]= %2d (%lu)\n",
           i, *a[i], (unsigned long) a[i],
           i, (*p)[i], (unsigned long) (*p + i));

  return 0;
```

```
}
```

Le istruzioni

```
a[0] = &a1;                    /* puntatori a var a1, ..., a5 */
a[1] = &a2;
a[2] = &a3;
a[3] = &a4;
a[4] = &a5;
```

assegnano agli elementi dell'array `a` di "dimensione 5 di tipo puntatore a tipo `int`" gli indirizzi di memoria delle variabili `a1`, ..., `a5` di tipo `int`. Le istruzioni

```
q = array;                     /* q punta ad array */
p = &q;                        /* p punta a q      */
```

assegnano alla variabile `q` di tipo "puntatore a tipo `int`" l'indirizzo di memoria dell'array `array` di dimensione `5` di tipo `int` ed alla variabile `p` di tipo "puntatore a puntatore a tipo `int`" l'indirizzo di memoria della variabile `q` puntatore a tipo `int`.

Se la dimensione del tipo `int` sul sistema è 4 quando il programma viene eseguito si avrà un output della forma

```
*a[0]=   1 (3221224076)    (*p)[0]= 10 (3221224088)
*a[1]=   2 (3221224080)    (*p)[1]= 20 (3221224092)
*a[2]=   3 (3221224084)    (*p)[2]= 30 (3221224096)
*a[3]=   4 (3221224108)    (*p)[3]= 40 (3221224100)
*a[4]=   5 (3221224112)    (*p)[4]= 50 (3221224104)
```

da cui risulta chiaramente che, indipendentemente dai valori numerici, `a` e `p` si riferiscono a due diverse organizzazioni in memoria. Infatti dalla terza colonna si nota che gli oggetti puntati da `a[0], a[1], a[2]` e `a[3], a[4]` occupano locazioni di memoria contigue, ma i due gruppi non sono contigui in memoria. L'ultima colonna mostra invece che gli oggetti puntati da `*p` occupano locazioni di memoria contigue.

Osserviamo che nonostante questa differenza sia `a` che `p` sono puntatori a locazioni di memoria che contengono un puntatore ad un oggetto di tipo `int` il cui valore è dato in entrambi i casi da `**a` o `**p`. Questo è il motivo per cui nel prototipo della dichiarazione e della definizione di funzione i parametri formali possono essere dichiarati indifferentemente sia come "array di puntatori a tipo `T`" o come "puntatore a puntatore a tipo `T`": quale che sia la dichiarazione utilizzata il parametro formale conterrà infatti in entrambi i casi un puntatore a puntatore a tipo `T`.

Nel corpo della funzione invece i parametri formali devono essere utilizzati, indipendentemente dalla forma della dichiarazione utilizzata, coerentemente con il tipo a cui si riferiscono altrimenti i risultati possono essere catastrofici. Questo vuol dire, ad esempio, che nella funzione `main()` il parametro formale `argv` deve essere sempre utilizzato come `*argv[i]` e non come `(*argv)[i]`, come mostra chiaramente il seguente programma

Programma: `comm_line-bad.c`

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

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

  printf("\n argc = %d\n\n", argc);

  printf(" Nome         -> argv[%d] = \"%s\"\n", 0, &(*argv)[0]);
  for (i = 1; i < argc; ++i) {
    printf(" Argomenti   -> argv[%d] = \"%s\"\n", i, &(*argv)[i]);
  }

  printf(" Terminatore -> argv[%d] = %s\n", i, &(*argv)[i]);
  printf("\n");

  return 0;
}
```

che quando viene eseguito invece di stampare il valore degli argomenti di linea produce

```
$ a.out argomenti di linea

 argc = 4

 Nome        -> argv[0] = "a.out"
 Argomenti   -> argv[1] = ".out"
 Argomenti   -> argv[2] = "out"
 Argomenti   -> argv[3] = "ut"
 Terminatore -> argv[4] = t
```

È infatti facile convincersi che con la scrittura `(*argv)[i]` ci si sposta lungo l'array di caratteri `argv[0]` e non da un array all'altro. Lo stesso risultato si sarebbe ottenuto definendo la funzione `main()` come

```
int main(int argc, char *argv[])
```

La "morale" è quindi che non sempre la dichiarazione dei parametri formali è sufficiente per determinare il tipo del parametro, per cui il suggerimento è quello di utilizzare dichiarazioni quanto più possibile aderenti al tipo effettivo del parametro.

2.34. Dichiarazione di tipo: `typedef` (Rev. 2.1)

Lo specificatore di classe di memorizzazione `typedef` permette di effettuare *dichiarazioni di tipo* per associare un identificatore ad un tipo. Quando in una dichiarazione viene specificata la classe di memorizzazione `typedef` l'identificatore che compare nella dichiarazione viene definito come l'identificatore di tipo (`typedef` *name*) del tipo che sarebbe stato associato all'identificatore se la dichiarazione fosse stata una dichiarazione usuale. Una volta definito l'identificatore di tipo può essere utilizzato ovunque possa essere utilizzato il tipo a cui fa riferimento. Le dichiarazioni di tipo non introducono nuovi tipi e gli identificatori di tipo sono considerati come *sinonimi* dei tipi a cui sono associati.

Consideriamo ad esempio le seguenti dichiarazioni:

```
char character;
int  meter, length;
```

che dichiarano la variabile `character` di tipo `char` e le variabili `meter` e `length` di tipo `int`. Se in queste dichiarazioni viene specificata la classe di memorizzazione `typedef` le dichiarazioni si trasformano nelle dichiarazioni di tipo:

```
typedef char character;
typedef int  meter, length;
```

che definiscono l'identificatore di tipo `character` come *sinonimo* del tipo `char` ed i due identificatori di tipo `meter` e `length` come *sinonimi* del tipo `int`. Questi identificatori possono essere usati ovunque sia possibile utilizzare il tipo `char` o il tipo `int`. Ad esempio la dichiarazione

```
character letter;
```

dichiara la variabile `letter` di tipo `character` mentre la dichiarazione

```
meter  distance;
```

dichiara la variabile `distance` di tipo `meter`. Siccome `character` è sinonimo del tipo `char` e `meter` del tipo `int` queste dichiarazioni sono a tutti gli effetti equivalenti alle dichiarazioni

```
char letter;
int  distance;
```

per cui a prima vista la dichiarazione di tipo del linguaggio C sembrerebbe l'analogo del comando `#define` del preprocessore C. Le precedenti dichiarazioni non sono infatti molto dissimili da

```
#define character char
#define meter int

character letter;
meter distance;
```

Nel caso di tipo semplici come possono essere il tipo `char` o il tipo `int` le due istruzioni sono effettivamente equivalenti, tuttavia la dichiarazione di tipo è molto più potente e versatile del comando `#define` perché premette di definire identificatori di tipo per tipi piuttosto complessi. Consideriamo ad esempio il seguente semplice programma:

```
int main (void)
{
 typedef int group[10];  /* group identificatore di tipo! */

 int     i;
 group var;              /* variabile di tipo group       */

 for (i=0; i < 10; ++i)
   var[i] = i;

 return 0;
}
```

In questo esempio l'istruzione

```
typedef int group[10];
```

definisce l'identificatore di tipo `group` come sinonimo del tipo "array di dimensione 10 di tipo `int`" mentre la successiva istruzione

```
group var;
```

dichiara la variabile `var` di tipo `group`. Di conseguenza siccome l'identificatore di tipo `group` è sinonimo del tipo "array di dimensione 10 di tipo `int`" la scrittura

```
for (i=0; i < 10; ++i)
  var[i] = i;
```

è perfettamente lecita. Un'operazione del genere è chiaramente di difficile, se non impossibile, realizzazione attraverso il preprocessore C.

Altri esempi di `typedef` *names* sono

```
typedef int group[10];  /* group tipo array dim 10 tipo int   */
typedef int *ip;        /* ip    tipo puntatore a int         */
typedef int (*fip)();   /* fip   tipo puntatore a funzione    */
                        /*         tipo  int                  */

ip       i_p;     /* i_p   puntatore a int                     */
ip   f_ptr();     /* f_ptr funzione tipo puntatore a int       */
fip     fi_p;     /* fi_p  puntatore a funzione tipo int       */
group   a[5];     /* a     array dimensione [10][5] tipo int   */
group *a10_p;     /* a5_p  puntatore ad array dimensione [10]  */
                  /*         tipo int                          */
```

Nel C99 lo specificatore di classe di memorizzazione `typedef` può essere usato anche con arrays di dimensione variabile. In questo caso la dimensione dell'array è quella nota al momento della dichiarazione di tipo e non quella eventualmente nota quando il `typedef` *name* è usato per dichiarare un array.

`typedef` *names* non possono essere combinati con altri specificatori di tipo. La seguente dichiarazione

```
typedef long int long_int;
unsigned long_int i;          /* Illegale */
```

è quindi illegale. È possibile tuttavia combinare qualificatori e `typedef` *names*, la dichiarazione

```
const long_int i;            /* Legale */
```

è di conseguenza perfettamente legale.

`typedef` *names* possono essere utilizzati in altre dichiarazioni di tipo. Ad esempio le istruzioni

```
typedef double type_d;
typedef type_d a5[5];

a5 array;
```

dichiarano la variabile `array` di tipo "array di dimensione 5 di tipo `type_d`" ovvero, essendo l'identificatore di tipo `type_d` un sinonimo del tipo `double`, "array di dimensione 5 di tipo `double`".

Lo specificatore di classe di memorizzazione `typedef` può essere utilizzato anche con le dichiarazioni di funzione ma il `typedef` *name* che ne risulta può essere utilizzato solo per dichiarare puntatori a funzioni, arrays di puntatori a funzioni e così via ma non per definire o dichiarare funzioni. La dichiarazione di tipo

```
typedef double dble_func();
```

definisce l'identificatore di tipo `dble_func` come sinonimo del tipo "funzione di tipo `double`". Questo può essere usato ad esempio per dichiarare puntatori a funzioni di tipo `double`

```
dble_func *fd_p;
```

o arrays di puntatori a funzioni di tipo `double`

```
dble_func *fd_p[7];
```

ma non può essere usato per definire una funzione poiché una definizione del tipo

```
dble_func func(double x)
{
  return (3.0 * x);
}
```

verrebbe interpretata come la definizione di una funzione che ritorna un'altra funzione. Non vi è modo di aggirare questo problema per cui la funzione va definita nel modo usuale come

```
double func(double x)
{
  return (3.0 * x);
}
```

come se l'identificatore di tipo `dble_func` non esistesse.

Lo Standard C permette di includere informazioni sui parametri formali delle funzioni nelle dichiarazioni di tipo per cui la dichiarazione di tipo

```
typedef double d_func_d(double x);
```

definisce l'identificatore di tipo `d_func_d` come sinonimo di "funzione di tipo `double` con un parametro di tipo `double`" mentre la dichiarazione

```
typedef unsigned int (*dbl_f_p)(int, double);
```

definisce l'identificatore di tipo `dbl_f_p` come sinonimo di "puntatore a funzione di tipo `unsigned int` con un parametro di tipo `int` ed un parametro di tipo `double`".

Le dichiarazioni effettuate con `typedef` *names* sottostanno alle usuali regole delle dichiarazioni per cui non devono dare origine a tipi non permessi dal linguaggio C, ad esempio la dichiarazione

```
dble_func array[10];
```

è illegale perché dichiara un array di funzioni.

Gli identificatori di tipo, come gli identificatori ordinari, possono essere ridefiniti all'interno dei blocchi:

```
typedef double type_d;
type_d var;
...
{
  int type_d;
  type_d = 4;
  ...
}
```

L'unica differenza è che nella ridefinizione dell'identificatore lo specificatore di tipo non può essere omesso con l'assunzione che in assenza di uno specificatore di tipo esplicito venga utilizzato di *default* il tipo `int`. È importante ricordarsi che i `type` *names* sono nella stessa classe di overloading degli identificatori delle variabili, delle funzioni e così via. Di conseguenza sebbene le dichiarazioni seguenti

```
typedef double dbl;
dbl dbl;
```

siano perfettamente lecite la seconda dichiarazione nasconde la prima ridefinendo l'identificatore `dbl` come identificatore di una variabile di tipo `double` cosicchè `dbl` non può più essere utilizzato come identificatore di tipo.

La possibilità di poter dichiarare identificatori di tipo ha diversi vantaggi. Il primo e più evidente è l'analogo del comando `#define`, ossia quello di poter utilizzare sinonimi che riflettano l'utilizzo degli oggetti e/o la loro natura. Il vantaggio di usare una dichiarazione di tipo invece che il comando `#define` del preprocessore risiede nel fatto che una dichiarazione di tipo è un'istruzione del linguaggio C per cui il compilatore ne controlla la correttezza della sintassi. Inoltre la dichiarazione di tipo permette di definire `types` *names* per tipi piuttosto complessi ed infatti, come vedremo più avanti, lo specificatore di classe di memorizzazione `typedef` trova largo uso con oggetti come strutture ed unioni.

L'utilizzo di identificatori di tipo non solo permette di abbreviare dichiarazioni di oggetti complessi migliorando così la facilità di lettura del programma, ma ne aumenta anche la portabilità perché permette di "nascondere" definizioni dipendenti dal sistema nei files di header. Un esempio è l'identificatore di tipo `size_t` definito nel file di header `stddef.h` che fornisce il tipo intero senza segno del valore dell'operatore `sizeof`. Molti sistemi usano per `size_t` il tipo `unsigned long int`, altri il tipo `unsigned int`. Alcuni sistemi infine usano ancora il tipo con segno `int`.

Il principale effetto collaterale della possibilità di utilizzare normali identificatori come sinonimi di tipo è che il significato delle istruzioni può dipendere dal contesto. Ad esempio se `f` è un identificatore di tipo l'istruzione

```
f (*x);
```

dichiara la variabile `x` di tipo "puntatore a `f`", in caso contrario l'istruzione è una chiamata alla funzione `f()` con argomento `*x`. Di conseguenza è una buona forma di programmazione

quella di utilizzare una convenzione per gli identificatori di tipo in modo da poterli distinguere facilmente dagli altri identificatori. Ad esempio molti identificatori di tipo di sistema finiscono con il suffisso "`_t`" come `size_t`.

2.35. Composizione di dichiaratori (Rev. 2.1)

Nel linguaggio C i dichiaratori di tipo possono essere combinati tra di loro in modo da formare tipi più complessi. Ogni combinazione è valida purché il tipo risultante non sia uno dei seguenti tipi vietati in C:

- Ogni tipo che includa il tipo `void`, con l'esclusione di "...funzione che ritorna `void`" o "puntatore a `void`" (Standard C).
- "Array di funzioni di...". Gli arrays possono contenere puntatori a funzioni ma non funzioni.
- "Funzione che restituisce un array di...". Le funzioni possono restituire puntatori ad arrays ma non arrays.
- "Funzione che restituisce una funzione di ...". Le funzioni possono restituire puntatori a funzioni ma non funzioni.

Abbiamo già incontrato esempi di dichiarazioni di tipo composte da più dichiaratori. Ad esempio la dichiarazione della variabile `a_ptr` di tipo array di dimensione 5 di tipo puntatore a `int`

```
int *a_ptr[5];
```

è composta dai dichiaratori di array e di puntatore:

```
        a_ptr[5]   ⇒  array di 5 elementi
     *(a_ptr[5])   ⇒  di tipo puntatore
int *(a_ptr[5])   ⇒  a tipo int
```

L'ordine dei dichiaratori nella dichiarazione è importante per cui se necessario si devono utilizzare le parentesi "`()`" per modificare la precedenza degli operatori e fornire il corretto ordine alla dichiarazione. Nella dichiarazione precedente le parentesi sono state omesse perché l'operatore di indice "`[]`" ha la precedenza sull'operatore di dereferenza "`*`" e quindi le parentesi sono superflue. Al contrario nella dichiarazione di una variabile di tipo puntatore ad array di tipo `int`

```
int (*a_p)[5];
```

le parentesi sono necessarie per fornire il corretto ordine alla dichiarazione, prima il dichiaratore di puntatore "`*`" e poi quello di array "`[]`", e quindi non possono essere omesse.

Lo Standard C richiede che vi possano essere almeno 12 livelli di parentesi nella dichiarazione, il che permette di costruire tipi piuttosto complessi. Tuttavia dal punto di vista di una buona

programmazione l'uso di tipi troppo complessi è sconsigliabile perché rende il programma di difficile lettura. Ad esempio la dichiarazione

```
double (*(*(*x)())[5])();
```

della variabile `x` di tipo "puntatore a funzione che ritorna un puntatore ad un array di 5 elementi ciascuno dei quali è un puntatore ad una funzione che ritorna `double`" non è certo delle più immediate. In questi casi è molto più conveniente dividere la dichiarazione in tipi più semplici e poi combinarli insieme. La dichiarazione della variabile `x` ad esempio può essere divisa come

```
typedef double (*fd_p)();      /* fd_p   tipo puntatore a funzione */
                               /*        che ritorna double        */
typedef fd_p   (*a5_ptr)[5];   /* a5_ptr tipo puntatore ad array   */
                               /*        di 5 elementi tipo fd_p   */
a5_ptr (*x)();                 /* x      puntatore a funzione che  */
                               /*        ritorna a5_ptr            */
```

a tutto vantaggio della chiarezza.

Come ulteriore esempio di tipi composti da più dichiaratori il seguente programma che integra l'equazione del moto di un pendolo con l'algoritmo di Eulero mostra l'uso del tipo "array di puntatori a funzione". Questo problema è già stato risolto utilizzando una funzione che integra un sistema di due equazioni differenziali ordinarie del primo ordine. Utilizzando degli arrays sia per le variabili che per le funzioni è possibile riscrivere il programma in forma più generale con una funzione che integra un sistema di equazioni differenziali ordinarie del primo ordine composto da un numero arbitrario `N` di equazioni, ottenendo così un programma utilizzabile per integrare equazioni del moto anche in dimensione superiore ad uno.

Programma: `euler-array_func.c`

```
/*
 * Descrizione :  Integrazione moto del pendolo con Eulero.
 *                Esempio uso array punatori a funzione
 *
 * $yaC-Primer: euler-array_func.c, v 1.1 27.02.2005 AC $
 */
#include <stdio.h>
#include <math.h>

#define N 2         /* # variabili */

/* Prototipi */
void euler(double h, double *x, double (*der[])(double *));
double x_dot(double *x);
double v_dot(double *x);

int main(void)
{
  int        stp;
  int    stp_max = 1000;       /* # max passi integrazione */
  double       h = 0.01;       /* passo di integrazione    */
```

```
  double    x[N];               /* variabili                */
  double (*der[N])(double *);   /* derivate variabili       */

  /* Condizione Iniziale */
  x[0]    = 1.0;                    /* x      */
  x[1]    = 0.0;                    /* v      */

  /* Funzioni dot(x) e dot(v) */
  der[0] = x_dot;                 /* dot(x) */
  der[1] = v_dot;                 /* dot(v) */

  /* integrazione */
  for (stp = 0; stp < stp_max; ++stp) {
    euler(h, x, der);
    printf("%.4f\t%.4f\t%.4f\n", stp * h, x[0], x[1]);
  }
  return 0;
}

void euler(double h, double x[], double (*f[])(double *))
{
  int          i;
  double x_n[N];       /* variabili temporanee */

  /* valori a t+h */
  for (i = 0; i < N; ++i) x_n[i] = x[i] + h * (*f[i])(x);

  /* Aggiornamento valori */
  for (i = 0; i < N; ++i) x[i]   = x_n[i];

  return;
}

double x_dot(double x[])
{
  return x[1];
}

double v_dot(double x[])
{
  return (-sin(x[0]));
}
```

La dichiarazione

```
double (*der[N])(double *);
```

dichiara l'array `der` di dimensione `N` di tipo "puntatore a funzione di tipo `double` con un parametro di tipo `double *`":

`der[N]`	⇒	array di dimensione `N`
`*der[N]`	⇒	di tipo puntatore

`(*der[N])(double *)`	⇒	a funzione con un parametro di tipo `double *`
`double (*der[N])(double *)`	⇒	di tipo `double`

Come nel caso degli arrays di puntatori anche in questo caso le parentesi sono necessarie.

La dichiarazione può essere fatta in forma più trasparente utilizzando un `typedef` *name* per il tipo "puntatore a funzione di tipo `double` con un argomento di tipo `double *`":

```
typedef double (*fun_d_dp_t)(double *);

fun_d_dp_t der[N];
```

L'identificatore di tipo `fun_d_dp_t` può essere utilizzato anche nei prototipi delle funzioni, purché questi siano specificati dopo la definizione di tipo. Ad esempio il prototipo della funzione `euler()` può essere scritto come

```
void euler(double h, double *x, fun_d_dp_t *der);
```

a tutto vantaggio della chiarezza. Nel programma si è scelto di non utilizzare un identificatore di tipo per mostrare esplicitamente la composizione dei dichiaratori, tuttavia in generale l'uso di `typedef` *names* è consigliato sopratutto per dichiarazioni che coinvolvono più dichiaratori.

L'array `der` permette di raggruppare insieme le equazione che definiscono il sistema di equazioni differenziali e di accedervi mediante un indice, di conseguenza ciascuna variabile del sistema è univocamente determinata dal un indice che va da `0` a `N-1`. Questo programma assume che le funzioni dipendano esplicitamente solo dalle variabili `x[N]` e non ad esempio dal tempo, non è tuttavia difficile modificarlo per aggiungere nella dichiarazione dell'array `der` gli argomenti necessari. In alternativa si può utilizzare la dichiarazione

```
double (*der[N])();
```

lasciando la lista dei parametri non specificata. In questo secondo caso però è bene ricordarsi che il compilatore non può controllare la coerenza del numero e del tipo dei parametri formali delle funzioni nelle diverse parti del programma.

A questo punto la funzione che integra un sistema di `N` equazioni differenziali ordinarie del primo ordine con l'algoritmo di Eulero si scrive facilmente utilizzando un indice `i` per individuare ciascuna variabile:

```
void euler(double h, double x[], double (*f[])(double *))
{
  int          i;
  double x_n[N];        /* variabili temporanee */

  /* valori a t+h */
  for (i = 0; i < N; ++i) x_n[i] = x[i] + h * (*f[i])(x);

  /* Aggiornamento valori */
  for (i = 0; i < N; ++i) x[i]   = x_n[i];

  return;
}
```

I parametri formali della funzione `euler()` sono il passo di integrazione `h`, l'array `x` con il valore delle `N` variabili e l'array `f` con il puntatori alle `N` funzioni che rappresentano i termini a destra del sistema di equazioni differenziali. L'array locale `x_n` è utilizzata come variabile temporanea di appoggio. Nello Standard C nella chiamata a funzione effettuata con un puntatore a funzione l'operatore di dereferenza "`*`" può essere omesso, di conseguenza la chiamata `(*f[i])(x)` può anche essere scritta più semplicemente come `f[i](x)`.

Nel caso dell'equazione del moto del pendolo

$$\begin{cases} \dot{x} &= v \\ \dot{v} &= -\sin(x) \end{cases}$$

servono solo due variabili, (x, v), per cui la macro `N` è definita come

```
#define N 2
```

Le due funzioni che costituiscono il sistema sono

```
double x_dot(double x[])
{
  return x[1];
}
```

per la variabile x e

```
double v_dot(double x[])
{
  return (-sin(x[0]));
}
```

per la variabile v. Il loro indirizzo di memoria viene assegnato agli elementi dell'array `der` con le istruzioni

```
der[0] = x_dot;                 /* dot(x) */
der[1] = v_dot;                 /* dot(v) */
```

Infine le istruzioni

```
x[0]    = 1.0;                    /* x      */
x[1]    = 0.0;                    /* v      */
```

assegnano il valore iniziale delle due variabili (x, v).

In questo esempio il programma `euler-array_func.c` è stato usato per integrare l'equazione del moto di un pendolo, ma è abbastanza evidente che con poche modifiche può essere utilizzato per integrare un sistema composto da un numero N qualsiasi di equazioni differenziali ordinarie del primo ordine.

2.36. Funzioni con un numero variabile di parametri (Rev. 2.1)

Il linguaggio C permette di definire funzioni che prendono un numero arbitrario di parametri o parametri di tipo arbitrario. Un esempio sono le funzioni di Input/Output come `printf()` o `scanf()`.

Nella dichiarazione e definizione di una funzione con un numero arbitrario di parametri, o parametri di tipo arbitrario, la parte variabile della lista dei parametri viene indicata nel prototipo della funzione con "`,...`" (una virgola e tre punti). Ad esempio la dichiarazione

```
type f_vararg(type_fixed parameter_fixed, ...);
```

dichiara la funzione `f_vararg()` che prende un parametro fissato di tipo `type_fixed` ed una serie variabile di parametri. La parte variabile della lista dei parametri deve sempre seguire la parte fissa della lista.

La definizione di una funzione con un numero arbitrario di parametri, o parametri di tipo arbitrario, è complicata dal fatto che i dettagli su come gli argomenti in una chiamata a funzione con un numero arbitrario di parametri sono passati alla funzione dipende dal sistema, con ovvie conseguenze sulla portabilità dei programmi. Per ovviare a questo problema lo Standard C fornisce una serie di *utilities* definite nel file di header `stdarg.h` che permette di standardizzare l'uso delle funzioni con un numero variabile di parametri. Il Traditional C forniva delle utilities analoghe definite nel file di header `varargs.h`. L'uso di `stdarg.h` differisce da quello di `varargs.h` per il fatto che lo Standard C richiede che vi sia almeno un parametro fissato prima della parte variabile della lista dei parametri, mentre nel Traditional C tutta la lista dei parametri doveva essere variabile.

2.36.1. Utilities `stdarg.h`

Le utilities fornite dal file di header `stdarg.h` sono:

- `typedef ... va_list;`

  Questo è il tipo dell'oggetto, nel seguito chiamato `ap`, utilizzato per scorrere gli argomenti della funzione della parte variabile della lista dei parametri. La dichiarazione dell'oggetto `ap` di tipo `va_list` è quindi

  ```
  va_list ap;
  ```

  Cosa sia veramente l'oggetto `ap` dipende dal sistema, tuttavia non è necessario saperlo ai fini del suo utilizzo.

- `#define va_start(va_list ap, type_last Last_Fixed_Param);`

  dove `type_last` è il tipo `T` dell'ultimo parametro fisso della funzione, ossia quello che precede i tre punti "`...`" nel prototipo della dichiarazione della funzione. Questa macro deve essere chiamata prima di accedere alla parte variabile della lista dei parametri della funzione con `var_arg` e `var_end` per per inizializzare `ap` in modo che punti al primo argomento della parte variabile della lista dei parametri. Nello Standard C `va_start` prende due parametri: l'oggetto `ap` di tipo `va_list` e l'ultimo parametro fisso `Last_Fixed_Param` della funzione. Ad esempio la chiamata

  ```
  int f(int param, ...);

  va_start(ap, param);
  ```

inizializza `ap` in modo che punti al primo parametro della parte variabile della lista dei parametri della funzione `f()`.

- `#define va_arg(va_list ap, type);`

Questa macro restituisce il valore dell'argomento della parte variabile della lista dei parametri indicato da `ap` ed aggiorna `ap` in modo che punti all'argomento successivo della lista, se questo esiste. La macro prende due argomenti: `ap` di tipo `va_list` ed il tipo `type` dell'argomento di cui restituire il valore. Il valore restituito è di tipo `type`. Il tipo `type` deve essere scritto in una forma tale che se gli viene aggiunto il suffisso "`*`" si ottiene correttamente il tipo "puntatore a `type`". La prima chiamata di `va_arg` restituisce il valore del primo argomento della parte variabile della lista dei parametri.

- `void va_end(va_list ap);`

Questa macro o funzione deve essere chiamata dopo che tutti gli argomenti della parte variabile della lista dei parametri sono stati letti con `var_arg` per effettuare le operazioni di "pulizia" e "chiusura" di `ap`.

- `void va_copy(va_list dest, va_list src);`

Questa macro introdotta dallo Standard C99 copia l'oggetto `src` di tipo `va_list` nell'oggetto `dest` di tipo `va_list`. Questo permette di avere due oggetti di tipo `va_list` che si riferiscono alla stessa lista di argomenti e quindi di rileggere eventualmente i valori degli argomenti.

La definizione di queste macros dipende dal sistema, di seguito ne è riportato un esempio

```
typedef char * va_list;

#define va_start(ap, last) \
        ((void) ((ap) = (va_list) &last + sizeof(last)))

#define va_arg(ap, type) \
        (*(type *)((ap) += sizeof(type), (ap) - sizeof(type)))

#define va_end(ap) \
        ((void) (ap = 0))

#define va_copy(dest, src) \
        ((void) ((dest) = (src)))
```

che assume che il valore degli argomenti della parte variabile della lista dei parametri sia memorizzato sequenzialmente il locazioni di memoria contigue a partire dal valore dell'ultimo parametro fisso della funzione. Per la macro `var_arg` si è usata una forma più esplicita al posto della forma più compatta, ma meno immediata,

```
#define va_arg(ap, type) \
        (*((type *) (ap))++)
```

Il seguente programma illustra l'uso di queste macro.

Programma: `test-func_stdarg.c`

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

double sum_d(int n, ...);

int main(void)
{
  double a = 1.0, b = 2.0, c = 3.0, d = 4.0;

  printf("a       = %f\n", sum_d(1, a));
  printf("a+b     = %f\n", sum_d(2, a, b));
  printf("a+b+c   = %f\n", sum_d(3, a, b, c));
  printf("a+b+c+d = %f\n", sum_d(4, a, b, c, d));

  return 0;
}

double sum_d(int n, ...)
{
  int    arg;          /* contatore argomenti */
  double sum;          /* somma argomenti     */
  va_list ap;          /* lista argomenti     */

  va_start(ap, n);     /* Inizializzazione lista argomenti */

  sum = 0.0;
  for (arg = 0; arg < n; ++arg) {
    sum += va_arg(ap, double);
  }

  va_end(ap);          /* Chiusura lista argomenti */

  return sum;
}
```

In questo semplice esempio la funzione `sum_d()` prende un numero variabile di parametri di tipo `double` e ne restituisce la somma dei valori. La funzione ha un solo parametro fisso di tipo `int` il cui valore è uguale al numero di argomenti passati alla funzione oltre al parametro fisso.

La scrittura della funzione è piuttosto semplice. Per prima cosa viene dichiarata la variabile `ap` di tipo `va_list` per accedere alla lista degli argomenti:

```
va_list ap;          /* lista argomenti     */
```

La variabile `ap` viene poi inizializzata con la macro `va_start` indicando che la parte variabile della lista dei parametri inizia dopo il parametro fisso `n`

```
va_start(ap, n);     /* Inizializzazione lista argomenti */
```

Infine vengono letti con la macro `var_arg` i valori degli argomenti di tipo `double` e sommati

```
sum = 0.0;
for (arg = 0; arg < n; ++arg) {
  sum += va_arg(ap, double);
}
```

Prima di terminare l'esecuzione della funzione la lista viene chiusa con l'istruzione `va_end(ap)`.

Quando il programma viene eseguito si ha il seguente output

```
a       = 1.000000
a+b     = 3.000000
a+b+c   = 6.000000
a+b+c+d = 10.000000
```

Il seguente esempio mostra il caso in cui non solo il numero ma anche il tipo dei parametri può variare.

Programma: `test-func_stdarg-2.c`

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

#define INT   1
#define DBL   2

void print_df(int n, int *type, ...);

int main(void)
{
  int  type[4] = {INT, INT, DBL, DBL};

  print_df(4, type, 1, 2, 3.0, 4.0);

  return 0;
}

void print_df(int n, int *type, ...)
{
  int    arg;          /* contatore argomenti */
  va_list ap;          /* lista argomenti     */

  va_start(ap, type);    /* Inizializzazione lista argomenti */

  for (arg = 0; arg < n; ++arg) {
    if (type[arg] == INT) {
      printf("valore intero: %d\n", va_arg(ap, int));
    }
    else {
      printf("valore double: %f\n", va_arg(ap, double));
```

```
    }
  }

  va_end(ap);          /* Chiusura lista argomenti */

  return;
}
```

La funzione `print_df()` prende due parametri fissi `n` di tipo `int` il cui valore come nel caso precedente è uguale al numero di argomenti passati alla funzione oltre ai due fissi, ed un array di tipo `int` che contiene un "codice identificativo" del tipo dell'argomento. In questo caso si utilizzano le macros `INT` e `DBL` per indicare rispettivamente il tipo `int` e `double`.

La gestione della parte variabile della lista dei parametri non differisce molto dalla funzione `sum_d()` dell'esempio precedente, con la sola differenza che la variabile `ap` deve essere inizializzata con il secondo parametro fisso

```
va_start(ap, type);    /* Inizializzazione lista argomenti */
```

Quando il programma viene eseguito si ha

```
valore intero: 1
valore intero: 2
valore double: 3.000000
valore double: 4.000000
```

2.37. Allocazione dinamica della memoria (Rev. 2.0.3)

In tutti gli esempi discussi fino ad ora gli oggetti utilizzati, come ad esempio variabili od array, sono sempre stati creati, ed eventualmente distrutti, automaticamente dal compilatore secondo la loro definizione. In alcuni casi, tuttavia, può essere utile poter creare e/o modificare dinamicamente gli oggetti.

Ad esempio non sempre la dimensione di un array è nota in anticipo e quindi per aumentare la portabilità di un programma può essere conveniente lasciarla variabile. Questo problema è stato risolto con l'aiuto di *macros* ed il preprocessore C, come mostrato dalla seguente porzione di programma.

```
#define N 100

int main(void)
{
  int x[N];

  ......

  return (0);
}
```

Quando il programma viene compilato il preprocessore C sostituisce alla *macro* `N` la stringa di caratteri "`100`" prima di passare le linee di programma al compilatore vero e proprio cosicchè quest'ultimo riceve il programma:

```
int main(void)
{
  int x[100];

  ......

  return (0);
}
```

per cui `x` è un array di `100` elementi di tipo `int`. Per variare la dimensione dell'array `x` basta cambiare la definizione della macro `N` e ricompilare il programma.

Questa soluzione ha l'evidente svantaggio di richiedere la compilazione del programma ogni qual volta si debba cambiare la dimensione dell'array, cosa che tuttavia potrebbe non essere troppo fastidiosa in quanto le macros possono essere definite direttamente in compilazione mediante la *flag* "`-D`" del compilatore C senza quindi dovere ricorrere ad un "*editing*" del file.

La limitazione maggiore nel ricorrere al preprocessore C è che non sempre è possibile utilizzare *macros* per assegnare spazio di memoria per gli oggetti che si vogliono utilizzare. Ad esempio la dimensione dell'array potrebbe dipendere dal contesto e quindi variare dinamicamente durante l'esecuzione del programma, ovvero si ha a che fare con oggetti più complessi di semplici arrays come strutture ed unioni, che vedremo tra poco.

Il linguaggio C premette di allocare, ossia di *assegnare*, dinamicamente della memoria mediante le funzioni

```
#include <stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

i cui prototipi si trovano nel file di *header* `stdlib.h`.

Le funzioni `malloc()`, `calloc()` e `realloc()` restituiscono un puntatore di tipo "*generico*" `void *` perchè la memoria allocata deve poter contenere ogni tipo di oggetto.

2.37.1. Funzione `malloc()`

La funzione `malloc()` (*memory allocation*) è la funzione principale di questa classe, ed il suo prototipo, è:

```
void *malloc(size_t size);
```

La funzione `malloc()` prende un parametro di tipo `size_t` (*size type*) che in Standard C è un tipo intero privo di segno, di solito `unsigned int`, definito per comodità nel file di header `stdlib.h` ed, in Standard C, anche in `stddef.h`. Il tipo `size_t` è un esempio di tipo derivato

ottenuto mediante l'uso dell'istruzione `typedef`:

```
typedef unsigned int size_t;
```

La funzione `malloc()` assegna uno spazio di memoria, in una zona di memoria chiamata *heap*, sufficiente per contenere un oggetto la cui dimensione, come misurata dall'operatore `sizeof`, è `size`. Se l'allocazione ha successo la funzione `malloc()` restituisce un puntatore di tipo `void *` al primo elemento di memoria dello spazio di memoria assegnato. Il puntatore può essere convertito, mediante un *cast*, in un puntatore ad un oggetto di tipo diverso. Se per qualche motivo l'assegnazione fallisce viene restituito il puntatore nullo `NULL`. Se il valore di *size* è *0* nello Standard C `malloc()` può restituire sia il puntatore nullo che uno non-nullo, in entrambi i casi il puntatore non deve essere utilizzato.

Lo spazio di memoria allocato dalla funzione `malloc()` non viene inizializzato.

Come esempio di utilizzo della funzione `malloc()` consideriamo l'allocazione di uno spazio di memoria per un array di tipo `int`. Questa prende la forma:

```
#include <stdlib.h>

int main(void)
{
  int *array;                          /* puntatore all'array   */
  unsigned int n;                      /* dimensione dell'array */
  ......
  array = malloc(n * sizeof(int));     /* spazio per n int      */
  if (array == NULL) {
      fprintf(stderr, "Out of memory\n");
      exit(1);
  }
  ......
  return (0);
}
```

La funzione `malloc()` restituisce un puntatore di tipo `void *` per cui l'utilizzo del cast esplicito:

```
array = (int *) malloc(n * sizeof(int));
```

non è strettamente necessario in quanto lo Standard C richiede che nell'assegnazione il tipo `void *` venga convertito implicitamente al tipo del puntatore a sinistra dell'assegnazione. Il suo uso è tuttavia una buona pratica di programmazione poichè, ad esempio, rende l'individuazione di eventuali errori più semplice. Nel Traditional C il cast esplicito è invece necessario.

Un'altra buona pratica di programmazione è il controllo del valore restituito da `malloc()`

```
if (array == NULL) { ...
```

Questo, tuttavia, viene spesso omesso assumendo di avere memoria a sufficienza. Il risultato è che il programma può andare in "crash" inaspettatamente se per qualche motivo la memoria non risulti disponibile.

Infine l'uso dell'operatore `sizeof()` rende il codice più portabile e facilmente utilizzabile su computers con differenti dimensioni per il tipo `int`.

2.37.2. Funzione `calloc()`

La funzione `malloc()` non inizializza la memoria allocata, per cui se nell'esempio precedente è necessario inizializzare a zero l'array questo va fatto esplicitamente. Alternativamente si può usare la funzione `calloc()` (*contiguous allocation*) il cui prototipo è:

```
void *calloc(size_t nmemb, size_t size);
```

La funzione `calloc()` alloca uno spazio di memoria contiguo per un'array di `nmemb` elementi ognuno di dimensione, come misurata dall'operatore `typedef`, `size`. La memoria allocata viene inizializzata *bit-a-bit* ponendo tutti i suoi bits uguali a `0`. Se l'allocazione ha successo, `calloc()` restituisce un puntatore di tipo `void *` al primo elemento di memoria dello spazio di memoria allocato, in caso contrario viene restituito il puntatore nullo `NULL`. Se `nmeb` o `size` sono nulli `calloc()` ha un comportamento analogo a quello `malloc()` per `size` nullo.

Se nell'esempio precedente l'array `x` dovesse essere inizializzato a zero avremmo potuto scrivere:

```
array = calloc(n,sizeof(int));
```

ovvero

```
array = (int *) calloc(n,sizeof(int));
```

Osserviamo tuttavia che porre tutti bits della memoria allocata uguali a `0` non necessariamente significa che gli elementi dell'array sono inizializzati con il valore `0`, ad esempio nella rappresentazione *floating-point* lo `0` potrebbe avere una codifica differente da una semplice sequenza di zeri.

> **Nota**
> La funzione `malloc()` non azzera la memoria ed è quindi più veloce della funzione `calloc()`, per cui se non è necessario inizializzare bit-a-bit a zero la memoria l'uso di `malloc()` è preferibile a quello di `calloc()`, specialmente se si richiedono grandi quantità di memoria.

2.37.3. Funzione `realloc()`

La funzione `realloc()`, il cui prototipo è:

```
void *realloc(void *ptr, size_t size);
```

prende come argomento un puntatore `ptr` ad una zona di memoria precedentemente allocata con la funzione `malloc()` o `calloc()` e ne cambia la sua dimensione a `size` conservandone il contenuto. Se necessario il contenuto è copiato in una nuova regione di memoria di dimensione `size`. La funzione ritorna il puntatore di tipo `void *` al primo elemento della (nuova) zona di

memoria allocata. Se la richiesta non può essere soddisfatta `realloc()` restituisce il puntatore nullo `NULL` e la memoria originale non viene toccata. Se `ptr` è un puntatore nullo `realloc()` si comporta come `malloc()`. Se invece `ptr` è un puntatore valido ma `size` è `0`, allora `realloc()` restituisce un puntatore nullo o non-nullo, come `malloc()`, e la zona di memoria puntata da `ptr` viene deallocata ed il suo contenuto viene perso.

Se la nuova dimensione richiesta è più piccola di quella originale, allora parte del contenuto alla fine della zona di memoria originale viene perso. Se invece la nuova dimensione è più grande di quella originale, allora il contenuto della memoria viene preservato e nuovo spazio viene allocato alla fine della zona di memoria originale. Lo spazio aggiunto non viene inizializzato.

Se il puntatore restituito da `realloc()` differisce da `ptr`, si deve assumere che la vecchia zona di memoria è stata deallocata e quindi non deve essere utilizzata.

Il seguente esempio illustra il tipico uso della funzione `realloc()` per espandere la dimensione di un array. La funzione `add_sample()` aggiunge il nuovo valore alla fine dell'array indicata dal puntatore `array` aumentando, se necessario, la dimensione e ritorna il numero di valori inseriti.

Funzione: `add_sample.c`

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

#define SIZE_INC 10         /* incremento dimensione array */
int add_sample(int value, int **array, int *size)
{
  static int sample = 0;       /* contatore elementi inseriti */

  ++sample;

  if (sample > *size) {          /* se necessario aumenta la */
    int *new_array;              /* dimensione dell'array    */
    unsigned int  new_size;

    new_size = *size + SIZE_INC;
    new_array = (int *) realloc(*array, new_size * sizeof(int));

    if (new_array == NULL) {
      fprintf(stderr, "Cannot reallocate\n");
      return -1;
    }
    else {
      *array = new_array;        /* (nuova) locazione di memoria */
      *size  = new_size;         /* nuova dimensione dell'array  */
    }
  }

 (*array)[sample-1] = value;

  return sample;                 /* valore della funzione */
}
```

```
#undef INC_SIZE
```

Per una migliore efficienza la dimensione dell'array viene aumentata di `SIZE_INC` e non di `1`.

2.37.4. Funzione `free()`

Quando la memoria allocata con una delle precedenti funzioni non è più necessaria può essere deallocata, ossia restituita all'*heap*, mediante la funzione `free()` il cui prototipo è:

```
void free(void *ptr);
```

La funzione `free()` dealloca la memoria puntata dal puntatore `ptr`, che deve essere un puntatore ad una zona di memoria preventivamente allocata da `malloc()`, `calloc()` o `realloc()`. In caso contrario, o se `free(ptr)` è già stato chiamato, il risultato è impredicibile. Se `ptr` è il puntatore nullo `NULL` non viene effettuata nessuna operazione. Una volta deallocata la memoria non deve essere utilizzata poichè un suo eventuale utilizzo genera risultati impredicibili.

Le seguenti linee di programma illustrano l'utilizzo delle funzioni `malloc()` e `free()` per allocare lo spazio per un array di tipo `float` all'interno di una funzione.

```
void do_something(int n)
{
  float *array;      /* puntatore all'array */

  array = malloc(n * sizeof(float));
  if (array == NULL) {
      fprintf(stderr, "Out of memory\n");
      exit(1);
  }
  ......

  free(array);    /* dealloca lo spazio usato */
  array = NULL;   /* azzera il puntatore      */

  return;
}
```

Restituire all'*heap* la memoria alla fine della funzione potrebbe sembrare un'operazione superflua, ma non lo è. La memoria allocata da `malloc()`, `calloc()` e `realloc()` non viene deallocata automaticamente, per cui se questa non viene deallocata prima della fine della funzione `do_something` la zona di memoria associata ad `array` viene persa! Infatti quando termina l'esecuzione della funzione `do_something` la variabile `array` scompare essendo una variabile con scopo locale e classe temporanea. Tuttavia per il sistema la memoria che era associata ad `array` risulta ancore usata e quindi non disponibile, anche se non vi è più un puntatore associato ad essa (*memory leakage*).

Se la funzione `do_something` viene chiamata più volte senza restituire lo spazio di memoria allocato, ad ogni chiamata viene allocata una quantità di memoria di dimensione `n *`

`sizeof(float)`, per cui se questa viene chiamata molte volte il programma può facilmente esaurire tutta la memoria disponibile ed andare in "crash".

Utilizzare una zona di memoria dopo che è stata deallocata è come utilizzare un array con un indice fuori dai limiti (*out-of-bounds error*). Cosa succede effettivamente dipende da cosa vi è nelle locazioni di memoria in questione, ed il programma può produrre risultati inaspettati. Per questo, sebbene non sia richiesto, è buona norma azzerare i puntatori alla memoria deallocata assegnandogli il valore `NULL`, come mostrato nell'esempio.

Il seguente programma illustra l'utilizzo dell'allocazione dinamica della memoria creando dinamicamente un array, inizializzandola con dei numeri aleatori e stampandone infine gli elementi e la loro media.

Esempio: `dynamic_array.c`

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

void   init_darray(double *a, int n);
void   print_darray(double *a, int n);
double mean_darray(double *a, int n);

int main(void)
{
  int n;                    /* dimensione dell'array          */
  double *array;            /* puntatore all'array            */
  char line[81];


  printf("\n%s","Dimensione array [< 1 exit]: " );
  fgets(line, sizeof(line), stdin);
  sscanf(line,"%d",&n);

  if ( n < 1 ) {
    printf ("\nBye!\n");
    return 1;
  }

  array = (double *) malloc(n * sizeof(double));
  if (array == NULL) {
    fprintf(stderr,"Out of memory\n");
    exit(1);
  }

  init_darray(array,n);              /* Inizializza l'array */

  printf("\n\tArray\n\n");
  print_darray(array,n);             /* stampa gli elementi */

  printf("\n\t Mean: %f\n", mean_darray(array,n));

  free(array);                      /* dealloca la memoria   */
```

```
  array = NULL;                         /* e azzera il puntatore */

  printf("\nBye!\n");
  return 0;
 }

void init_darray(double *a, int n)
{
  int seed = 3479;          /* seme generatore numeri random */
  int i;

  srand(seed);
  for (i = 0; i < n; ++i) {
    a[i] = (double) rand() / (RAND_MAX + 1.0);
  }
  return;
}

void print_darray(double *a, int n)
{
  int i;

  for (i = 0; i < n; ++i) {
    printf("array[%d] = %f\n", i, *(a++));
  }
  return;
}

double mean_darray(double *a, int n)
{
  int i;
  double sum = 0;
  for (i = 0; i < n; ++i) {
    sum += *(a++);
  }
  return ( sum / (double) n );
}
```

Quando il programma viene eseguito si ottiene

```
 Dimensione array [< 1 exit]: 10

          Array

 array[0] = 0.778810
 array[1] = 0.065634
 array[2] = 0.385730
 array[3] = 0.127350
 array[4] = 0.456630
 array[5] = 0.217571
 array[6] = 0.957286
 array[7] = 0.854427
```

```
array[8] = 0.784371
array[9] = 0.541195

         Mean: 0.516900

Bye!
```

Il programma utilizza la funzione `rand()`, il cui prototipo è

```
int rand(void);
```

che restituisce un numero intero aleatorio con distribuzione uniforme tra `0` e `RAND_MAX`. Di conseguenza l'espressione

```
(double) rand() / (RAND_MAX + 1.0)
```

genera un numero aleatorio con distribuzione uniforme tra `0` (incluso) e `1` (escluso). Il generatore `rand()` deve essere *inizializzato* con la funzione

```
void srand(unsigned int seed);
```

dove `seed` è un numero dato, `3479` nell'esempio. Le funzioni `rand()` e `srand()`, come il valore di `RAND_MAX` sono definite nella libreria di sistema `stdlib.h`.

2.38. Esempio: Istogramma in frequenza con allocazione dinamica della memoria (Rev. 2.0.2)

Questo programma calcola l'istogramma di frequenza di un set di dati x_i letti da un file. Il programma si basa sul programma `isto1.c` già discusso ma, a differenza di quest'ultimo, non richiede di specificare a priori il numero massimo di bins dell'istogramma.

Come fatto nel programma `isto1.c`, per ridurre la quantità di memoria necessaria, i dati vengono prima letti per determinarne il valore massimo $x_{\max}$ e minimo $x_{\min}$, necessari per definire il supporto dell'istogramma, ed numero totale di dati n_{data}. Poi, dopo aver "riavvolto" il file, si rileggono tutti i dati dal file inserendoli di volta in volta nel "bin" corrispondente di un istogramma su n_{bin} "bins".

Il programma prende come input il numero `n_bin` di bins dell'istogramma, il nome del file contenente i dati e il nome del file di output su cui scrivere l'istogramma. Se quest'ultimo non è specificato, l'output è sullo `stdout`.

Programma: `isto_dyn.c`

```
/*****************************************************************

- Descrizione : Calcola l'istogramma in frequenza di un set di
                dati letti da in file.

- Input :       numero di bin
                file di dati
```

```
                file con l'istogramma [se non dato --> stdout]

- Output:       istogramma se non specificato file output

- $Id: isto_dyn.c v 3.1 08.11.04 AC
*****************************************************************/
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

/* Funzioni Private */
double *dvect(int n);
FILE   *open_file(const char *path, const char *mode);
void    usage(const char *strn);

int main(int argc, char *argv[])
{
  int n_data, ib, n_bin;                  /* Contatori */

  double x_tmp;                           /* Dato di input       */
  double *xb, *yb;                        /* bin dell'istogramma */
  double x_min, x_max;                    /* min max dati        */
  double dx;                              /* ampiezza bin        */
  double dy;                              /* Normalizzazione bin */
  FILE   *in_stream;
  FILE   *out_stream;

  /* Controlla i dati di input sono sufficienti */
  if (argc < 3) usage(argv[0]);

  /* Array per l'istogramma*/
  n_bin = atoi(argv[1]);
  xb    = dvect(n_bin);
  yb    = dvect(n_bin);

  /* File di input */
  in_stream = open_file(argv[2],"r");

  /* Leggi i dati e calcola il massimo e minimo */
  if (fscanf(in_stream,"%lf", &x_tmp) == EOF) {
    fprintf(stderr,"Errore in lettura del file...\n\n");
    exit (1);
  }

  n_data = 1;
  x_min  = x_tmp;
  x_max  = x_min;
  while (fscanf(in_stream,"%lf", &x_tmp) != EOF ) {
    ++n_data;
    if ( x_tmp > x_max ) {
      x_max = x_tmp;
    }
```

```
    else if ( x_tmp < x_min ) {
      x_min = x_tmp;
    }
  }

  /* calcola l'istogramma di frequenza */

  /* Largezza e normalizzazione bin */
  dx = 1.0001 * (x_max - x_min) / ((double) n_bin);
  dy = 1.0 / ((double) n_data);

  /* Inizializzazione */
  for(ib = 0; ib < n_bin; ib++) {
    xb[ib] = x_min + dx * (0.5 + (double) ib );
    yb[ib] = 0.00;
  }

  /* conta i dati per bin */
  rewind(in_stream);
  while (fscanf(in_stream,"%lf", &x_tmp) != EOF ) {
    ib      = (int) ( (x_tmp - x_min) / dx );
    yb[ib] += dy;
  }
  fclose(in_stream);

  /* Ora scrivi l'istogramma */

  /* File di output, se non specificato stdout */
  if (argc < 4 ) {
    out_stream = stdout;
  }
  else {
    out_stream = open_file(argv[3],"w");
  }

  fprintf(out_stream, "# Letti %d dati dal file %s\n",
          n_data, argv[2]);
  fprintf(out_stream, "# x_min %.3f \t x_max: %.3f \t dx: %.3f\n",
          x_min, x_max, dx);

  for (ib = 0; ib < n_bin; ++ib) {
    fprintf(out_stream, "%f \t %f\n", xb[ib], yb[ib]);
  }

  /* Libera la memoria */
  free(xb);
  free(yb);

  fclose (out_stream);
  return (0);
}
```

```
/* ============================== */
/* Funzioni Private               */
/* ============================== */

/* ----
 * usage()
 *
 * Stampa le istruzioni d'uso ed esce
 */
void usage(const char *strn)
{
  printf("\n Calcola l'istogramma normalizzato su #bins per \n");
  printf(" dati {x} contenuti nel file filename.\n");
  printf(" Uso del programma: \n\n");
  printf("   %s #bin inputfile [outfile] \n\n", strn);
  exit (1);
  return;
}

/* ----
 * dvect()
 *
 * Alloca la memoria per un array di n oggetti
 * di tipo double
 */
double *dvect(int n)
{
  double *v;

  v = (double *) malloc( n * sizeof(double));
  if (v == NULL) {
    fprintf(stderr,"\n Cannot allocate memory \n");
    exit(EXIT_FAILURE);
  }
  return v;
}

/* ----
 * open_file()
 *
 * apre un file e controlla che non ci siano errori
 */
FILE *open_file(const char *path, const char *mode)
{
  FILE *fa;

  if ( (fa = fopen(path, mode)) == NULL)
  {
    fprintf(stderr,"\n\nErrore in apertura di %s\n", path);
    exit(EXIT_FAILURE);
  }
  return fa;
```

```
}
```

Note sul programma

- `int main(int argc, char *argv[])`

Il programma legge i dati di input direttamente dalla linea di comando. Quindi se lo compiliamo con il comando

```
$ cc isto_dyn.c -o isto
```

il programma va eseguito come

```
$ isto par_1 par_2 ...
```

- `if (argc < 3) usage(argv[0]);`

Controlla che il numero di parametri passati sia sufficiente, in caso contrario chiama la funzione `usage()` che stampa un messaggio esplicativo. La funzione `usage()` prende come parametro `argv[0]` che contiene il nome del comando. Se il programma viene eseguito come all'esempio del punto precedente `argv[0]` contiene "`isto`".

- `n_bin = atoi(argv[1]);`

La funzione `atoi()` trasforma la stringa `argv[1]`, che contiene il numero di bins voluto come stringa di caratteri, in un valore di tipo `int`. Il prototipo della funzione `atoi()` è:

```
#include <stdlib.h>

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

done `ntpr` è il puntatore alla stringa da convertire.

- ```
  xb    = dvect(n_bin);
  yb    = dvect(n_bin);
  ```

La funzione `dvect()` definita nel programma, riserva dinamicamente lo spazio per un array di tipo `double`. La funzione prende come parametro la dimensione dell'array ritorna il puntatore alla memoria allocata. In questo caso `xb` e `yb` sono due arrays di dimensione `n_bin` (primo elemento `[0]`, ultimo elemento `[n_bin-1]`). In caso di errore la funzione interrompe l'esecuzione. La funzione `dvect()` usa la funzione `malloc()`, per cui gli arrays non sono inizializzati.

- `in_stream = open_file(argv[2],"r");`

il nome del file di input è dato dal secondo argomento passato.
  ```

- ```
  if (argc < 4 ) {
    out_stream = stdout;
  }
  else {
    out_stream = open_file(argv[3],"w");
  }
  ```

  Se il nome del file di output, terzo argomento passato, non è fornitp allora l'output viene inviato sullo `stdout`. Questa istruzione poteva essere anche scritta usando l'operatore condizionale "`?:`"

  ```
  out_stream = (argc < 4) ? stdout : open_file(argv[3],"w");
  ```

- ```
  free(xb);
  free(yb);
  ```

  Libera la memoria usata. Queste istruzioni non sono strettamente necessarie perchè il programma termina.

- ```
  double *dvect(int n)
  {
    double *v;

    v = (double *) malloc( n * sizeof(double));
    if (v == NULL) {
      fprintf(stderr,"\n␣Cannot␣allocate␣memory␣\n");
      exit(EXIT_FAILURE);
    }
    return v;
  }
  ```

  Riserva la memoria per un array di tipo `double` di dimensione `n`. Se vi è un errore nell'allocazione della memoria l'esecuzione del programma viene interrotta ritornando il valore `EXIT_FAILURE`. La macros `EXIT_FAILURE` è definita nel file di header `stdlib.h` usualmente come `1`.

Uso del programma

Supponiamo il file `isto.dat` contenga 1000 dati e che noi ne vogliamo l'istogramma su `10` bins. Questo si ottiene con il comando:

```
$ isto 10 isto.dat
# Letti 1000 dati dal file isto.dat
# x_min 35.857   x_max: 66.298   dx: 3.044
37.379202         0.017000
40.423607         0.039000
43.468011         0.088000
46.512415         0.178000
49.556820         0.256000
52.601224         0.203000
55.645629         0.139000
```
```

```
58.690033       0.059000
61.734437       0.016000
64.778842       0.005000
```

Se invece si usa il comando

```
$ isto 10 isto.dat isto.his
```

l'istogramma viene scritto nel file `isto.his`.

Infine se non si fornisce il numero sufficiente di parametri il programma scrive il messaggio:

```
$ isto

 Calcola l'istogramma normalizzato su #bins per
 dati {x} contenuti nel file filename.
 Uso del programma:

   isto #bin inputfile [outfile]
```

Esercizi

1. Modificare il programma in modo che vengano considerati per l'istogramma solo i dati nell'intervallo $[x_{\min}, x_{\max}]$ specificato.

2. Utilizzando il Teorema del Limite Centrale, scrivere un semplice programma che generi numeri aleatori con distribuzione Gaussiana di media nulla e varianza 1.

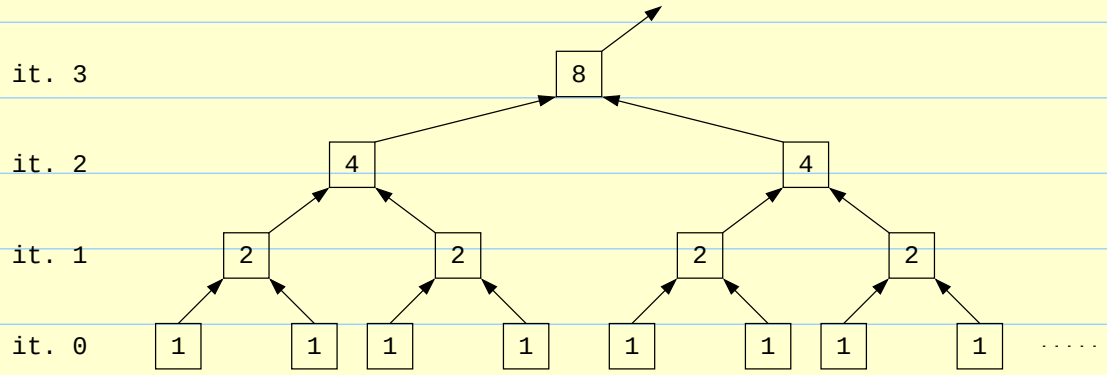
2.39. Esempio: algoritmo mergesort per l'ordinamento di un array

(Rev. 2.0.1)

Un metodo molto efficiente per ordinare una serie di dati secondo un ordine prestabilito è quello di procedere per ordinamenti successivi di sottosequenze di dati. Per illustrare il metodo supponiamo di dover ordinare in modo crescente una serie di numeri contenuti in un array di dimensione $n_{\text{data}} = 2^N$. In questo caso l'algoritmo è relativamente semplice:

- Si ordina ogni gruppo formato da 2 elementi successivi;
- Si ordina ogni gruppo formata da 4 elementi successivi;
- Si ordina ogni gruppo formata da 8 elementi successivi;

e così via fino ad ordinare tutta la sequenza. Se indichiamo con un intero $1, 2, 3 \ldots$ il numero di elementi in ogni gruppo, il procedimento può essere facilmente visualizzato su un albero binario:



Più si sale lungo l'albero più la dimensione dei gruppi ordinati aumenta. Il numero di iterazioni richieste è chiaramente N, ossia pari al logaritmo del numero di dati da ordinare.

Il passaggio da un livello dell'albero al successivo richiede di unire (*merging*) insieme due gruppi per ottenerne uno nuovo di dimensione doppia. Sfruttando il fatto che i due gruppi sono ordinati il nuovo gruppo può essere costruito direttamente ordinato semplicemente mettendo nel nuovo gruppo i dati presi in ordine dai due gruppi. Per illustrare il procedimento supponiamo che i due gruppi di partenza siano contenuti negli arrays `a` e `b` mentre il nuovo gruppo nell'array `c`. L'algoritmo di "fusione" ordinata procede allora come segue:

- $\min(a[0], b[0]) \rightarrow c[0]$. Supponiamo $a[0] \rightarrow c[0]$.
- $\min(a[1], b[0]) \rightarrow c[1]$. Supponiamo $a[1] \rightarrow c[1]$.
- $\min(a[2], b[0]) \rightarrow c[2]$. Supponiamo $b[0] \rightarrow c[2]$.
- $\min(a[2], b[1]) \rightarrow c[3]$. Supponiamo $a[2] \rightarrow c[3]$.

e così via fino a quando sono stati inseriti tutti gli elementi dei due gruppi `a` e `b`.

La fusione dei due gruppi richiede un numero di operazioni proporzionale al numero di elementi da inserire. Di conseguenza, siccome il numero di iterazioni lungo l'albero cresce come il logaritmo del numero di dati da ordinare, il numero totale di operazioni, e quindi anche il tempo richiesto, cresce come $n_{\text{data}} \log n_{\text{data}}$. L'algoritmo di *mergesort* è quindi particolarmente efficiente.

La seguente funzione `mergesort_darray()` ordina un array di 2^N elementi di tipo `double` utilizzando l'algoritmo *mergesort* descritto.

Funzione: `mergesort_darray.c`

```
/*****************************************************************

- Descrizione : ordina un array di 2^N elementi di tipo double

- $Id: mergesort_darray.c v 1.1 21.10.03 AC
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
```

```c
/* Funzioni private al modulo */
void merge_darray(double *a, double *b, double *c, int na, int nb);

void mergesort_darray(double *array, int n)
{
  register int i;
  int          grp_size;   /* grp_size := dimensione gruppo */
  int          grp_pos;    /* grp_pos  := posizione gruppo   */
  double      *a_tmp;

  /* Riserva la memoria per un array temporaneo  */
  a_tmp = (double *) malloc(n * sizeof(double));
  if (a_tmp == NULL) {
    fprintf (stderr, "\n Array too long !!!!\n\n");
    exit (1);
  }

  for (grp_size = 1; grp_size < n; grp_size *= 2) {
    for (grp_pos = 0; grp_pos < n - grp_size;
                      grp_pos += 2 * grp_size) {
      merge_darray(array + grp_pos, array + grp_pos + grp_size,
                   a_tmp + grp_pos, grp_size, grp_size);
    }

    for (i = 0; i < n; ++i) {
      array[i] = a_tmp[i];
    }
  }

  /* Libera la memoria usata */
  free(a_tmp);

  return;
}

/* ---
 * merge_darray()
 *
 * unisce due arrays ordinate di dimensione na e nb
 * in un array ordinata di dimensione na+nb
 */
void merge_darray(double *a, double *b, double *c, int na, int nb)
{
  int ia = 0, ib = 0, ic = 0;

  while ( ia < na && ib < nb ) {
    c[ic++] = ( a[ia] < b[ib] ) ? a[ia++] : b[ib++];
  }

  /* Copia quanto rimane dall array a o b in c*/
  while ( ia < na ) c[ic++] = a[ia++];
```

```
  while ( ib < nb ) c[ic++] = b[ib++];
  return;
}
```

Note su mergesort_darray.c

- ```
  a_tmp = (double *) malloc(n * sizeof(double));
  if (a_tmp == NULL) {
    fprintf (stderr, "\n Array too long !!!!\n\n");
    exit (1);
  }
  ```

  La funzione usa un array temporaneo di dimensione uguale a quella da ordinare su cui copiare i dati per passare al livello superiore dell'albero.

- ```
  for (grp_size = 1; grp_size < n; grp_size *= 2) {
      .....
  }
  ```

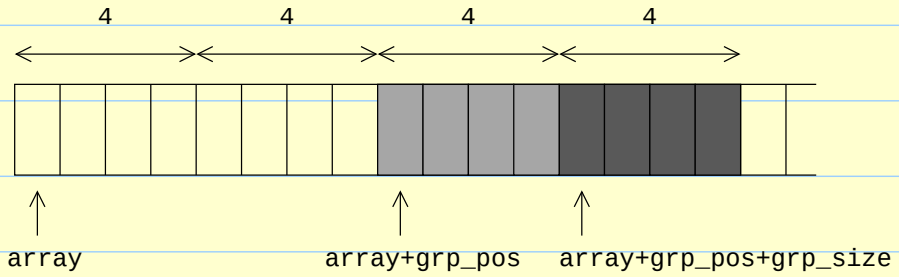
  Ciclo sui livelli dell'albero. La variabile `grp_size` è la dimensione dei gruppi ad un dato livello dell'albero che vale `grp_size=1` per il livello più basso con gruppi di dimensione 1 e raddoppia ogni volta che si sale di livello. Il ciclo si ferma quando si è raggiunta la cima dell'albero.

- ```
  for (grp_pos = 0; grp_pos < n - grp_size;
                    grp_pos += 2 * grp_size) {
      .....
  }
  ```

  Ciclo sulle coppie dei gruppi da unire per salire di un livello nell'albero. La variabile `grp_pos` indica la posizione relativa al primo elemento dell'array (*offset*) del primo elemento del primo dei due gruppi di dimensione `grp_size` da unire. Per passare da una coppia ad un'altra la variabile deve essere incrementata del doppio della dimensione dei gruppi.

- ```
  merge_darray(array + grp_pos, array + grp_pos + grp_size,
               a_tmp + grp_pos, grp_size, grp_size);
  ```

  La funzione `merge_darray()` unisce due arrays ordinate in una terza array ordinata di dimensione uguale alla somma delle dimensioni degli arrays originari, e quindi fa passare da un livello dell'albero al successivo. I due gruppi da unire sono quello che inizia all'elemento nell'array `array+grp_pos` ed il successivo che inizia all'elemento nell'array `array+grp_pos+grp_size`, come mostrato nella figura seguente nel caso di `grp_size=4`.



Il risultato dell'unione dei due gruppi viene scritto nell'array temporanea `a_tmp` a partire dalla posizione `array+grp_pos`.

- ```
  for (i = 0; i < n; ++i) {
      array[i] = a_tmp[i];
  }
  ```

Una volta unite tutte le coppie di gruppi successivi di dimensione `grp_size` l'array temporanea `a_tmp` contiene l'array ordinata al livello successivo dell'albero. Questa viene quindi ricopiata sull'array prima di raddoppiare la dimensione dei gruppi e salire di un altro livello.

- ```
  free(a_tmp);
  ```

Prima di uscire dalla funzione libera la memoria usata.

- ```
  while ( ia < na && ib < nb ) {
      c[ic++] = ( a[ia] < b[ib] ) ? a[ia++] : b[ib++];
  }
  ```

Ciclo di unione ordinata degli arrays `a` e `b`. L'operatore condizionale "`?:`" ritorna il valore più piccolo tra quello contenuto nell'array `a` e l'array `b`, contemporaneamente viene incrementato di uno l'indice dell'array corrispondente. Il ciclo termina quando si sono esauriti tutti gli elementi di una delle due arrays.

- ```
  while ( ia < na ) c[ic++] = a[ia++];
  while ( ib < nb ) c[ic++] = b[ib++];
  ```

Copia al fondo dell'array `c` quanto resta dell'array che contiene ancora elementi.

Test della funzione `mergesort_darray()`

Il seguente programma crea un array di tipo `double` dimensione letta dallo `stdin`, la riempie con dei numeri aleatori e la ordina utilizzando la funzione `mergesort_darray()`. Gli arrays prima e dopo l'ordinamento vengono stampati sullo `stdout`.

Programma: `test-mergesort.c`

```
/*****************************************************************

- Descrizione : test di mergersort_darray()

- $Id: test-mergesort.c v 1.2 29.09.04 AC
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>

extern void mergesort_darray(double *array, int n);

void init_darray(double *a, int n);
void print_darray(double *a, int n);

int main(void)
{
  int n;
  double *array;
  char line[81];

  printf("\n%s","Dimensione array [< 1 exit]: " );
  fgets(line,sizeof(line), stdin);
  sscanf(line,"%d", &n);

  if ( n < 1 ) {
    printf ("\nYou are joking!\n");
    return 1;
  }

  array = (double *) malloc(n * sizeof(double));
  if (array == NULL) {
    fprintf (stderr, "\n Array too long !!!!\n\n");
    exit (1);
  }

  init_darray(array, n);
  printf("\n\tArray\n\n");
  print_darray(array,n);

  mergesort_darray(array, n);
  printf("\n\tSorted Array\n\n");
  print_darray(array,n);

  return 0;
}


void init_darray(double *a, int n)
{
  int seed = 3479;          /* seme generatore numeri random */
  int i;
```

```
  srand(seed);

  for (i = 0; i < n; ++i) {
    a[i] = rand() / (RAND_MAX + 1.0);
  }
  return;
}


void print_darray(double *a, int n)
{
  int i;

  for (i = 0; i < n; ++i) {
    printf("array[%d] = %f\n", i, *(a++));
  }
  return;
}
```

Quando il programma viene eseguito si ottiene

```
Dimensione array [< 1 exit]: 4

        Array

array[0] = 0.778810
array[1] = 0.065634
array[2] = 0.385730
array[3] = 0.127350

        Sorted Array

array[0] = 0.065634
array[1] = 0.127350
array[2] = 0.385730
array[3] = 0.778810
```

Esercizi

1. Modificare la funzione `mergesort_darray.c` per utilizzarla con arrays di dimensione qualsiasi.
   Suggerimento: ogni numero intero può essere scritto come somma di potenze di 2 (rappresentazione binaria). Per ordinare un array di dimensione arbitraria scomporla in arrays di dimensione uguale ad una potenza di 2, ordinarle separatamente e ricomporle usando la funzione `merge_darray()`.

2. Scrivere un programma che ordina un array scambiando l'ordine di due elementi successivi. L'operazione deve essere ripetuta fino a quando l'array non è completamente

ordinata. Confrontarne l'efficienza con quella del *mergesort* ordinando arrays di grande dimensione.

2.40. Arrays multidimensionali dinamiche (Rev. 2.0)

Il linguaggio C permette di definire arrays dinamici con un numero qualsiasi di indici. Per semplicità ci limiteremo al solo caso di arrays con due indici, ma risulterà chiaro che la procedura può essere facilmente estesa ad arrays con un numero di indici qualsiasi.

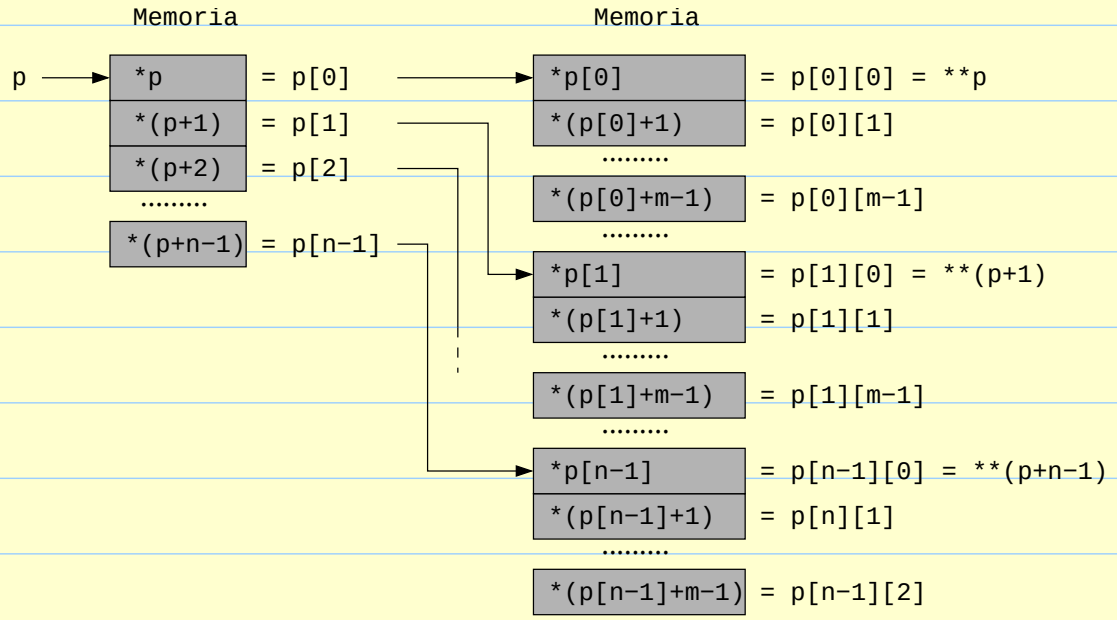
2.40.1. Creazione

Il modo più semplice di definire un array bidimensionale dinamico di `n`×`m` elementi di tipo `type` è quello di definire un array dinamico di `n` puntatori ad altrettanti arrays dinamici di `m` elementi di tipo `type` che rappresentano le righe (o colonne) dell'array. Ad esempio le seguenti istruzioni

```
int **p;
int   i;

p = (int **)malloc(n * sizeof(int *));
for (i = 0; i < n; ++i) {
  p[i] = (int *)malloc(n * sizeof(int));
}
```

definiscono l'array dinamico bidimensionale `p` di `n`×`m` elementi di tipo `int`. L'identificatore `p` è un "puntatore a puntatore" a tipo `int` poichè contiene l'indirizzo del primo elemento di un array di puntatori a tipo `int`, in particolare `*p=p[0]` è un puntatore alla prima riga (o colonna) dell'array mentre `p[0][0]` il contenuto dell'elemento "`[0][0]`" dell'array bidimensionale. L'organizzazione in memoria dell'array bidimensionale dinamico è mostrato nella figura seguente.



Osserviamo che gli arrays di dimensione `m` non occupano necessariamente unità di memoria contigue.

Da questa figura si nota anche che per accedere all'elemento `[i][j]` dell'array si può utilizzare la notazione usuale `p[i][j]`. Infatti l'elemento `[i][j]` è l'elemento `j`-esimo dell'`i`-esimo array di `m` elementi che rappresenta la riga (o colonna) dell'array bidimensionale. Di conseguenza per accedere all'elemento si deve

Selezionare `i`-esimo array ⇒ `p[i]`
Selezionare `j`-esimo elemento ⇒ `*(p[i]+j)`

⇒ `(p[i])[j] = p[i][j]`

Osserviamo che questa definizione segue molto da vicino l'interpretazione degli arrays bidimensionali usata nel linguaggio C. Infatti l'espressione

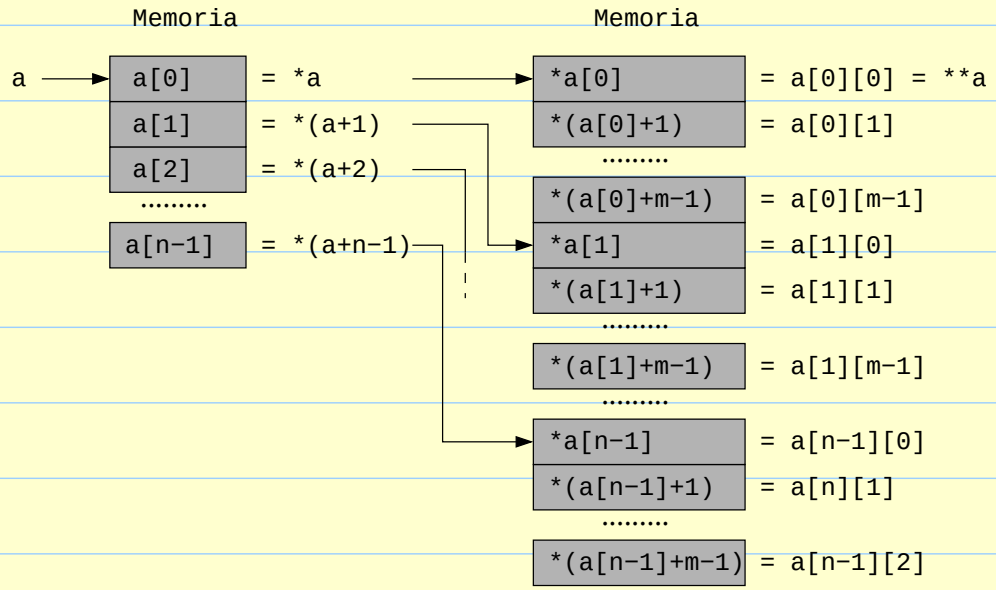
```
type a[n][m]
```

che definisce l'array bidimensionale `a` di `n`×`m` elementi di tipo `type` viene convertita automaticamente in

```
type (a[n])[m]
```

per cui l'identificatore `a` viene interpretato come "puntatore ad un array di `n` arrays di `m` elementi di tipo `type`".

Nonostante questa somiglianza l'array bidimensionale dinamico `p` non è *esattamente* equivalente all'array `a` a causa della differente organizzazione in memoria. Infatti `a` viene memorizzato come



e quindi contrariamente a `p` le sue righe (o colonne) occupano unità di memoria *contigue*.

Osserviamo infine che essendo `p` un puntatore a puntatore quando questo viene passato come argomento di funzione non si deve specificare la dimensione del secondo indice, cosa invece necessaria per l'array bidimensionale `a`.

2.40.2. Cancellazione

Per cancellare un array bidimensionale dinamico e restituire la memoria al sistema bisogna liberare lo spazio allocato per tutti gli arrays dinamici utilizzati. Di conseguenza per liberare lo spazio allocato per l'array bidimensionale `p` si devono utilizzare le seguenti istruzioni

```
int i;

for (i = 0; i < n; ++i) {
  free(p[i]);
}
free(p);
```

infatti se si libera solo lo spazio allocato a `p` quello allocato per i vari arrays di dimensione `m` non viene liberato ma viene perso non potendovi più accedere dopo aver cancellato l'array dinamico `p`.

2.41. Tipo Struttura (Rev. 2.0.9)

Supponiamo di dover scrivere un programma per gestire gli studenti di un corso e che ogni oggetto "studente" debba contenere le seguenti informazioni:

Nome	→	Stringa 60 caratteri
Cognome	→	Stringa 60 caratteri
Matricola	→	Stringa 10 caratteri
Voto	→	Numero intero

Chiaramente non possiamo usare un array poichè i dati sono di tipo diverso, interi e stringhe, mentre in un array tutti gli elementi devono essere necessariamente dello stesso tipo. Per aggirare questo problema è possibile utilizzare più arrays definendo un array per ogni tipo di informazione da memorizzare. Si avrà quindi un array per il nome, una per il cognome, una per la matricola e così via. Questa soluzione presenta tuttavia ovvie limitazioni.

In questi casi, ossia quando si devono trattare insiemi di dati di tipo diverso, conviene utilizzare il tipo di dati chiamato *struttura*. Il tipo *struttura* infatti non è altro che una collezione di oggetti che possono essere anche di tipo diverso. Ciascun oggetto rappresenta un elemento o campo (*field*) della struttura ed è individuato da un nome e non, come nel caso degli arrays, dalla sua posizione nella struttura. Le strutture permettono quindi di raggruppare facilmente insieme oggetti collegati fra di loro.

2.41.1. Dichiarazione del tipo struttura

La dichiarazione del tipo struttura è della forma:

```
struct structure-tag {
  field-type field-name;      /* comment */
  field-type field-name;      /* comment */
  . . . .
} variable-name;
```

dove

`structure-tag`	→	identificatore o nome del tipo struttura
`field-type`	→	tipo del campo della struttura
`field-name`	→	nome del campo della struttura
`variable-name`	→	nome della variabile di tipo `struct structure-tag`

Le dichiarazioni racchiuse nel blocco delimitato dalle parentesi graffe "`{...}`" definiscono i campi della struttura. Ogni istruzione

```
field-type field-name;
```

è una dichiarazione di variabile che definisce il nome ed il tipo del campo della struttura.

Il nome della struttura, dei suoi campi e delle variabili di tipo struttura hanno ognuno una propria classe di memorizzazione, per cui si può utilizzare lo stesso identificatore per la struttura, un suo campo ed una variabile di tipo struttura. I nomi dei campi di una stessa struttura devono invece essere diversi tra loro ma strutture differenti possono avere campi con lo stesso nome. Questo non crea confusione poichè i campi sono sempre associati alla struttura di appartenenza.

Nell'esempio degli studenti la definizione della struttura è:

```
struct student {
     char first_name[61];       /* Nome      */
     char  last_name[61];       /* Cognome   */
     char     std_id[11];       /* Matricola */
     int           grade;       /* Voto      */
} student_entry;
```

Analizzando più in dettaglio questa dichiarazione osserviamo che è composta di due parti.

La prima è la dichiarazione

```
struct student { ... }
```

che definisce il tipo di dati `struct student` la cui struttura è data dalle istruzioni contenute nel blocco racchiuso dalle parentesi graffe "{}". Siccome la struttura del nuovo tipo è completamente definita è possibile dichiarare variabili (od *oggetti*) di tipo `struct student` con l'usuale sintassi di dichiarazione di variabili. Ad esempio l'istruzione:

```
struct student studente_corso;
```

dichiara la variabile (od *oggetto*) `studente_corso` di tipo `struct student`.

La seconda parte della dichiarazione

```
struct student { ... } student_entry;
```

è la dichiarazione della variable `student_entry` di tipo `struct student`.

È possibile omettere sia la prima che la seconda parte nella dichiarazione della struttura omettendo rispettivamente `structure-tag` o `variable-name`.

Se si omette `structure-tag`, come in

```
struct {
     char first_name[61];       /* Nome      */
     char  last_name[61];       /* Cognome   */
     char     std_id[11];       /* Matricola */
     int           grade;       /* Voto      */
} student_entry;
```

si utilizza solo la *seconda* parte della dichiarazione, per cui in questo caso si dichiara la variabile `student_entry` di tipo `struct` con la struttura specificata nelle parentesi graffe, ma non un tipo di dati. Di conseguenza se si necessita di altre variabili di questo tipo queste vanno dichiarate esplicitamente allo stesso modo di `student_entry`.

Se invece si omette `variable-name`, come in

```
struct student {
     char first_name[61];       /* Nome      */
     char  last_name[61];       /* Cognome   */
     char     std_id[11];       /* Matricola */
     int           grade;       /* Voto      */
};
```

si utilizza solo la *prima* parte della dichiarazione della struttura. In questo caso si definisce il tipo di dati `struct student` con la struttura specificata nelle parentesi graffe, ma non

vengono dichiarate variabili di questo tipo. Queste devono essere dichiarate separatamente come:

```
struct student student_entry;
struct student studente_corso;
```

Infine è possibile omettere sia `structure-tag` che `variable-name`, come in:

```
struct {
     char first_name[61];      /* Nome      */
     char  last_name[61];      /* Cognome   */
     char     std_id[11];      /* Matricola */
     int           grade;      /* Voto      */
};
```

ottenendo così un pezzo di codice ~~sintatticamente corretto~~ ma ~~*perfettamente inutile*~~!

Spesso nella dichiarazione di strutture si utilizza l'istruzione typedef, come nell'esempio seguente:

```
struct complex_struct {
     double re;  /* real part of a complex number      */
     double im;  /* imaginary part of a complex number */
};

typedef struct complex_struct complex;
```

in cui si definisce il tipo `complex`, equivalente al tipo `struct complex_struct`. È quindi possibile dichiarare variabili che possono contenere dei numeri complessi come:

```
complex var_z;
```

Associare il nome `structure-tag` al tipo `struct` è considerata una buona pratica di programmazione tuttavia quando si usa `typedef` per definire un tipo il nome è spesso superfluo per cui in questi casi il più delle volte `structure-tag` viene omesso. Ad esempio il tipo `complex` dell'esempio precedente può essere definito anche come:

```
typedef struct {
     double re;  /* real part of a complex number      */
     double im;  /* imaginary part of a complex number */
} complex;
```

Lo scopo del tipo struttura è limitato al blocco che contiene la definizione per cui una struttura, sia che sia una variabile che una dichiarazione di tipo, è utilizzabile sono nel blocco in cui è definita ed in quelli in esso contenuti (se non nascosta) ma non al di fuori di esso.

Se nella dichiarazione di una struttura `structure-tag` è presente la dichiarazione *nasconde* esplicitamente ogni definizione precedente dell'*identificatore* `structure-tag`.

2.41.2. Accesso ai campi di una struttura

Per accedere ai campi di una struttura si utilizza l'operatore di selezione "`.`" la cui sintassi è:

```
structure.field-name
```

dove `structure` è un'espressione di tipo struttura e `field-name` il nome del campo del tipo struttura.

Ad esempio per modificare (o assegnare) il voto ad uno studente del corso useremo l'istruzione:

```
student_entry.grade = 18;
```

oppure per calcolare il modulo quadro di un numero complesso:

```
square_mod = var_z.re*var_z.re + var_z.im*var_z.im;
```

In questi due esempi `structure` è l'identificatore di due variabili di tipo `struct`, ma in generale può essere una qualsiasi espressione il cui valore sia di tipo struttura, come ad esempio una funzione.

Esempio: `func_struct.c`

```c
#include <stdio.h>

struct strc {
  int a, b;       /* campo a e b della struttura */
};

struct strc func(int);      /* funzione che ritorna una struttura */

int main(void)
{
  int i;
  struct strc x;


  x = func(1);
  printf("a: %d \t b: %d\n", x.a, x.b);

  x.a = func(-1).a;                          /* cambia campo a */
  printf("a: %d \t b: %d\n", x.a, x.b);

  i = func(-2).b;
  printf("i: %d\n", i);

  return 0;
}

struct strc func(int n)
{
  struct strc x;

  x.a = 1 * n;
  x.b = 2 * n;

  return x;
}
```

Quando questo programma viene eseguito si ha

```
a: 1      b: 2
a: -1     b: 2
i: -4
```

Ovviamente un'istruzione del tipo

```
func(2).a = 3;
```

è errata, anche se `func(2).a` è sintatticamente corretta, perchè assegna un valore ad una funzione.

Per accedere ai campi di una struttura attraverso un puntatore ad essa il linguaggio C fornisce l'operatore di selezione "`->`", ottenuto premendo successivamente il carattere "`-`" ed il carattere "`>`". La sintassi è simile a quella dell'operatore di selezione "`.`":

```
structure->field-name
```

dove `structure` è un'espressione di tipo "puntatore a tipo struttura" e `field-name` il nome del campo del tipo struttura. Diremo di più su questo operatore più avanti.

2.41.3. Dichiarazione ed inizializzazione di una variabile di tipo struttura

Variabili di tipo `struct` possono essere inizializzate al momento della dichiarazione mettendo la lista dei valori dei campi tra parentesi graffe "{}". Ad esempio nel caso degli studenti del corso potremo inizializzare un elemento come:

```
struct student {
        char first_name[61];      /* Nome      */
        char  last_name[61];      /* Cognome   */
        char     std_id[11];      /* Matricola */
        int           grade;      /* Voto      */
} student_entry_1 = {
        "Mario",                  /* Nome      */
        "Rossi",                  /* Cognome   */
        "0113567",                /* Matricola */
        20                        /* Voto      */
};
```

ovvero, se il tipo `struct student` è stato già definito,

```
struct student student_entry_2 = {
        "Giovanni",               /* Nome      */
        "Bianchi",                /* Cognome   */
        "0113565",                /* Matricola */
        18                        /* Voto      */
};
```

2.41.4. Campi

I campi di una struttura possono essere oggetti di tipo qualsiasi eccetto "funzione che ritorna ..." e `void`. Inoltre una struttura non può contenere la struttura stessa. Ad esempio la dichiarazione

```
struct list {
   int a;
   struct list b;          /* non valido */
   .....
};
```

non è permessa nonostante il tipo `struct list` sia definito *prima* della dichiarazione del campo `b`. Il motivo è che sebbene la definizione del tipo `struct list` inizi prima di essere utilizzato per definire il campo `b` questa non è finita al momento della sua utilizzazione:

```
struct list {  <-- inizio definzione
   int a;
   struct list b;
   .....
};  <-- fine definizione
```

Di conseguenza nella dichiarazione `struct list b` il tipo `struct list` è *incompleto* ed il compilatore non può quindi determinare le dimensioni del campo `b`.

Definizioni *incomplete* sono ammesse solo quando non sia richiesta la conoscenza della dimensione della struttura. Una struttura può quindi contenere puntatori a se stessa, come nella dichiarazione

```
struct list {
   int a;
   struct list *b;          /* OK */
   .....
};
```

od ad altre strutture la cui definizione non sia completa a patto che questa venga completata prima dell'utilizzo:

```
struct list {
   struct element *b;
   .....
};
struct element {
   int a;
   .....
};
```

Osserviamo che una costruzione di questo tipo può risultare pericolosa perchè la definizione di `struct element` nasconde una eventuale definizione precedente dell'identificatore `element` come tipo struttura, per cui se l'identificatore `element` è già associato ad un tipo struttura definito prima della definizione di `struct list` quest'ultima avrà un campo differente da quello voluto. Per ovviare a questi inconvenienti il linguaggio C permette di utilizzare una dichiarazione di tipo struttura *incompleta* composta dal solo identificatore

```
struct structure_tag;
```

per nascondere qualsiasi definizione precedente dell'identificatore `structure-tag`. Per cui il modo corretto di scrivere le precedenti definizioni è:

```
struct element;
struct list {
   struct element *b;
   .....
};
struct element {
   int a;
   .....
};
```

in modo da nascondere con la dichiarazione incompleta "`struct element;`" qualsiasi precedente definizione di `element`.

I nomi dei campi di una struttura sono definiti in una classe di memorizzazione associata alla struttura. Di conseguenza i campi di una stessa struttura devono avere nomi diversi, ma non necessariamente diversi da quelli di un'altra struttura o di variabili o di funzioni. Ad esempio nelle seguenti istruzioni:

```
int x;
struct a {int x; double y} y;
struct b {int y; double x} z;
```

l'identificatore `x` ha tre differenti e non conflittuali dichiarazioni: nella prima è una variabile di tipo `int`, nella seconda un campo di tipo `int` della struttura `struct a` ed infine nella terza un campo di tipo `double` della struttura `struct b`. Non vi sono problemi di confusione in quanto nelle espressioni le tre diverse dichiarazioni sono usate rispettivamente come:

```
x   y.x   z.x
```

I campi delle strutture possono a loro volta essere strutture purchè la definizione del tipo sia *completo* al momento della dichiarazione del campo. A questo proposito è bene tenere presente che se una struttura è definita in uno dei campi di una struttura il suo scopo è limitato alla parte di programma che va dalla sua definizione fino alla fine del blocco che contiene la definizione della struttura più esterna. Ad esempio nella seguente dichiarazione

```
struct a {
   struct b {int x, double y} z;
   .....
} d;
```

lo scopo, e quindi la validità, della struttura `struct b` è dalla sua dichiarazione fino alla fine del blocco che definisce la struttura `struct a`. Questo vuol dire che non è possibile dichiarare, o usare, oggetti di tipo `struct b` al di fuori della struttura `struct a` per cui ad esempio un'istruzione del tipo

```
struct b w;
```

fuori del blocco che definisce `struct a`, o in questo prima della definizione di `struct b`, non è permessa.

Osserviamo infine che per accedere ai campi della struttura `z` bisogna utilizzare due volte l'operatore di selezione "`.`" per cui ad esempio il campo `x` viene selezionato come:

```
d.z.x
```

L'uso delle parentesi non è necessario in quanto l'operatore di selezione "`.`" ha associatività a sinistra per cui `d.z.x` viene correttamente interpretato come

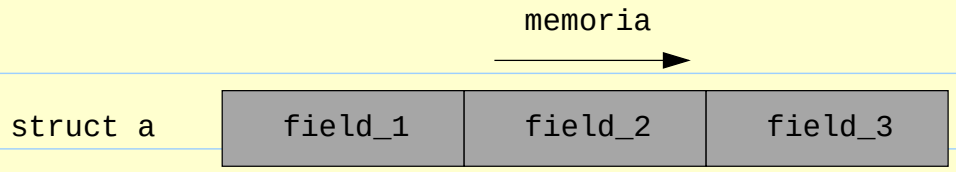
```
((d).z).x
```

Ossia: prendi la struttura `d`, di questa prendi il campo `z`, questo è a sua volta una struttura di cui prendi il campo `x`.

2.41.5. Ordinamento e dimensione di una struttura

Lo Standard C richiede che i campi di una struttura siano ordinati in memoria secondo la loro sequenza nella definizione della struttura e che il primo campo inizi all'indirizzo di memoria dove inizia la struttura, come mostrato nella seguente figura:

```
struct a {
   type_1 field_1;
   type_2 field_2;
   type_3 field_3;
};
```



I campi sono allocati nella memoria in ordine di indirizzo di memoria crescente con il primo campo posto all'inizio della struttura. Questo vuol dire che sono ordinati in memoria o da sinistra a destra se il computer usa l'ordinamento *big-endian* per i bytes o da destra a sinistra se invece utilizza l'ordinamento *little-endian*.

L'ordinamento in memoria dei campi del tipo struttura dipende solo dall'ordine con cui questi sono definiti nella definizione del tipo struttura e non dal modo come sono definiti. Di conseguenza i campi della struttura

```
struct a {
   int x;
   int y;
};
```

e quelli della struttura

```
struct a {
   int x, y;
};
```

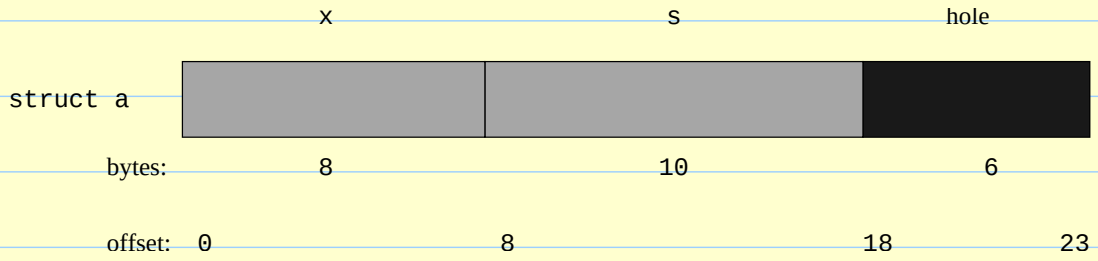
hanno lo stesso ordinamento in memoria.

Se i campi sono ordinati in memoria secondo il loro ordine nella definizione della struttura, non necessariamente questi sono contigui. Infatti tra la zona di memoria di un campo e quella del successivo vi possono essere delle locazioni inutilizzate, chiamate *holes* o *padding*, necessari per un corretto allineamento della struttura nella memoria.

Ad esempio se il computer richiede che gli oggetti di tipo `double` siano memorizzati un locazioni di memoria con un indirizzo multiplo di 8 bytes la struttura

```
struct a {
   double      x;
   char    s[10];
};
```

potrà essere ordinata in memoria come

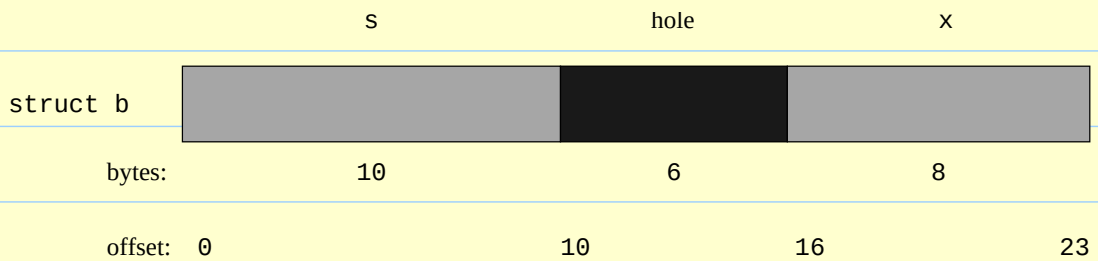


I 6 bytes aggiunti alla fine (*hole*) servono a far si che la dimensione della struttura sia un multiplo di 8 in modo tale che se si definisce un array di strutture di tipo `struct a` il campo `x` di tipo `double` si verrà sempre a trovare in locazioni di memoria di indirizzo multiplo di 8 bytes. Senza l'inserimento di un *hole* questo non sarebbe stato possibile.

In questo caso il *padding* è stato effettuato al fondo della struttura, ma in altri casi può essere necessario inserire degli *holes* tra i campi. Consideriamo ad esempio la struttura

```
struct b {
   char    s[10];
   double      x;
};
```

Questa contiene campi dello stesso tipo di quelli della struttura `struct a` ma l'ordine è diverso. Se di nuovo richiediamo che gli oggetti di tipo `double` siano allineati in modo che il loro indirizzo di memoria sia multiplo di 8 bytes la struttura `struct b` dovrà essere allineata ad esempio come



in modo che l'indirizzo del campo `x` sia sempre un multiplo di 8 bytes.

In genere il valore dei *bits* in questi *holes* non è predicibile e può differire da una struttura ad un altra o per una stessa struttura variare nel tempo.

La *dimensione* di una struttura è data dalla quantità di memoria necessaria per rappresentare correttamente tutti tipi dei suoi campi, inclusa quella lo occupata dagli *holes*. Di conseguenza la dimensione di una struttura è maggiore od uguale alla somma delle dimensioni dei suoi campi. Per conoscere la dimensione di una struttura si può utilizzare l'operatore `sizeof` in quanto lo spazio occupato dagli *holes* è contato nel valore ritornato dall'operatore.

2.41.6. Operazioni permesse e non

Il tipo struttura è un insieme di oggetti di diversa natura per cui una volta selezionato un campo, mediante l'operatore di selezione "`.`" o "`->`", è possibile effettuare con questo tutte le operazioni permesse con il tipo del campo.

Per quanto riguarda le strutture come tali non solo è possibile passare come parametri di funzioni puntatori a strutture e definire funzioni che ritornano "puntatori a strutture", ma le strutture stesse possono essere passate come parametri di funzioni e restituite da funzioni, come mostrato chiaramente dall'esempio precedente. Infine è anche possibile assegnarle, ad esempio

```
struct a {int x; int y;} z, w;
w = z;
```

assegna al campo `w.x` il valore del campo `z.x` ed al campo `w.y` il valore del campo `z.y`. Di conseguenza l'assegnazione `w = z` è a tutti gli effetti equivalente alle istruzioni

```
w.x = z.x;
w.y = z.y;
```

Ovviamente entrambi i membri dell'assegnazione devono essere strutture dello stesso tipo per cui, ad esempio, la seguente assegnazione

```
struct a {int x;       int y;} z;
struct b {double x; double y;} w;
w = z;              /* Non valida se w e z strutture diverse */
```

non è valida, anche se si effettua un cast esplicito

```
w = (struct b) z;   /* Non valida se w e z strutture diverse */
```

Non è invece possibile effettuare un test di uguaglianza diretto tra due strutture. Una struttura è una sequenza di campi di tipo diverso per cui l'unico modo di effettuare un test di uguaglianza che non richieda informazioni sulla natura dei campi sarebbe quella di confrontare bit-a-bit il contenuto della memoria occupata dalle due strutture. Tuttavia questa soluzione non è praticabile in quanto per motivi di allineamento in memoria una struttura può contenere *holes* tra un campo e l'altro. Queste sono zone di memoria il cui contenuto è arbitrario e che quindi vanificano qualsiasi test basato su un confronto bit-a-bit. Il test di uguaglianza tra strutture deve quindi basarsi sul confronto campo-per-campo. Il linguaggio C non fornisce

un confronto di questo tipo per cui se necessario questo deve essere scritto direttamente dal programmatore.

2.41.7. Bit Fields

Il linguaggio C permette di memorizzare i campi tipo intero delle strutture utilizzando un numero di bits inferiore a quello usualmente utilizzato per i tipi interi. Campi di questo tipo sono chiamati *bit fields* e sono specificati aggiungendo alla dichiarazione del campo il carattere due punti ":" seguito da un intero che indica da dimensione in bit del campo

```
struct structure-tag {
  integer-type field-name:width;   /* bit field dimensione width */
  . . . .
};
```

Ad esempio la seguente struttura

```
struct s {
  unsigned int a:4;      /* 4 bits */
  signed int   b:5;      /* 5 bits */
  int          c:5;      /* 5 bits */

};
```

ha tre campi di tipo bit field: il campo `a` che utilizza `4` bits ed i campi `b` e `c` che utilizzano entrambi `5` bits.

I campi di tipo bit field possono essere sia di tipo `unsigned int` che `signed int` o semplicemente `int`. Questi vengono chiamati rispettivamente *unsigned*, *signed* e *plain* bit fields. Siccome i campi di tipo bit field contengono tipi interi un bit field di n bits può rappresentare numeri interi da 0 a 2^n-1 se di tipo `unsigned` ovvero da -2^{n-1} a $2^{n-1}-1$, in rappresentazione a complemento a due, se di tipo `signed`. Un campo di tipo plain bit field può rappresentare a seconda del sistema sia numeri interi con segno che senza segno. Di conseguenza su un sistema che usa la rappresentazione a complemento a due le seguenti istruzioni

```
struct s v    = { -1, -1, -1};
int      i_us = v.a;
int      i_sg = v.b;
int      i_pl = v.c;
```

assegnano il valore `15` alla variabile `i_us` ed il valore `-1` alla variabile `i_sg`. Il valore assegnato alla variabile `i_pl` è invece `31` o `-1` a seconda della rappresentazione utilizzata dal sistema per il plain bit field. Questa segue le stesse regole della rappresentazione del tipo *plain* carattere `char`.

I campi di tipo bit field permettono una gestione molto dettagliata del numero di bits utilizzati per memorizzare i dati per cui sono spesso utilizzati per rappresentare strutture di dati con dimensioni fissate, ad esempio supponiamo che un particolare *device* accetti dati sotto forma di una sequenza di n bits in cui i primi n_1 rappresentino un tipo di informazione, i secondi n_2 un altro tipo e così via. Chiaramente la struttura dei dati adatta per questo device può

essere rappresentata da una struttura i cui campi siano bit fields di dimensione n_1, n_2 e così via.

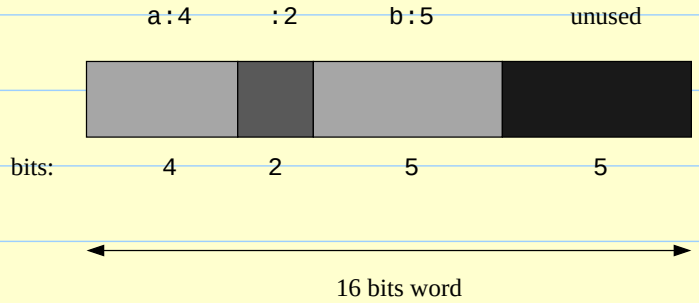
La limitazione principale nell'utilizzo dei campi di tipo bit field è dovuta al modo con cui i campi di una struttura, ed in particolare i bit fields, sono organizzati nella memoria riservata alla struttura. I campi di tipo bit field sono memorizzati in una struttura nel modo più compatto possibile, compatibile con le limitazioni imposte dall'allineamento, tuttavia l'organizzazione dipende dal sistema. Inoltre i compilatori possono fissare un numero massimo di bits utilizzabili per un bit field o limitazioni sul loro allineamento in memoria, normalmente legati alle dimensioni della parola del processore. Di conseguenza l'utilizzo di campi di tipo bit field spesso produce programmi non portabili e per questo i bit fields vengono utilizzati solo in quelle parti di programma specifiche del sistema.

È possibile introdurre dei campi con il ruolo di *holes* per forzare un particolare *padding* tra i campi semplicemente omettendo il nome del campo. Ad esempio se la dimensione della parola è di 16 bits la struttura

```
struct s {
  unsigned int a:4;      /* 4 bits */
  signed int    :2;      /* hole 2 bits */
  int           b:5;     /* 5 bits */

};
```

viene memorizzata come



Gli ultimi 5 bits della parola non vengono utilizzati.

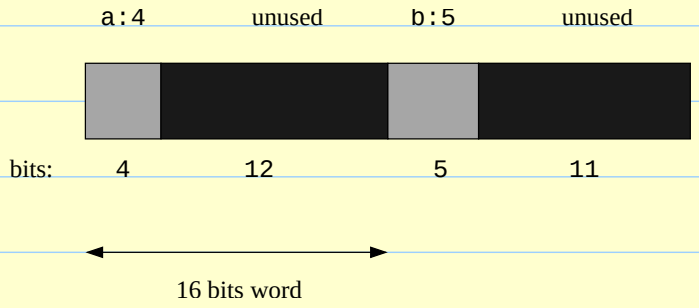
I campi di tipo bit field utilizzati per il *padding* non possono essere richiamati ed il contenuto è non predicibile e può variare nel tempo.

La lunghezza del campo utilizzato per il *padding* può essere uguale a `0` per indicare nella locazione di memoria dove è stato inserito l'ultimo bit field non se ne devono inserire altri. Gli eventuali bit fields successivi sono inseriti a partire dalla locazione di memoria successiva. Di conseguenza la struttura

```
struct s {
  unsigned int a:4;      /* 4 bits */
  signed int    :0;      /* hole 0 bits -> next memory location */
  int           b:5;     /* 5 bits */

};
```

viene memorizzata in un computer con parola di 16 bits come



Chiaramente non è possibile applicare l'operatore di referenza "`&`" ai campi di tipo bit field poichè la loro posizione in memoria non necessariamente coincide con una locazione di memoria indirizzabile come chiaramente mostrato negli esempi precedenti.

2.42. Tipo Unione (Rev. 2.0.3)

Il tipo *unione*, al pari del tipo struttura, è composto da una collezione di oggetti di diversa natura. La sintassi della dichiarazione del tipo unione è praticamente identica a quella della dichiarazione del tipo struttura:

```
union union-tag {
  field-type field-name;      /* comment */
  field-type field-name;      /* comment */
  . . . .
} variable-name;
```

dove

`union-tag`	:=	identificatore o nome del tipo unione
`field-type`	:=	tipo del campo della unione
`field-name`	:=	nome del campo della unione
`variable-name`	:=	nome della variabile di tipo `union union-tag`

Il tipo unione è per molti versi simile al tipo struttura per cui al tipo unione si applicano praticamente tutte le regole e le operazioni valide per il tipo struttura. Così ad esempio i campi di una unione possono essere di qualsiasi tipo incluso altre unioni, strutture o puntatori a strutture, ma non del tipo "funzione che ritorna ..." o `void`. Nello Standard C i campi delle unioni possono essere anche di tipo *bit fields* con le stesse regole valide per le strutture. Inoltre, come le strutture, una unione non può contenere l'unione stessa ma solo puntatori a se stessa. Per accedere direttamente ai campi si usa l'operatore di selezione "`.`" con la stessa sintassi utilizzata per le strutture, e così via.

La differenza tra il tipo struttura ed il tipo unione è nell'utilizzo della memoria: al tipo struttura viene assegnato uno spazio di memoria sufficiente a contenere tutti i campi con il primo campo allocato a partire dall'indirizzo di memoria dove inizia la struttura. Al contrario nel tipo unione tutti i campi sono allocati a partire dall'indirizzo di memoria dove inizia l'unione cosicchè una unione può contenere il valore di un solo campo alla volta. La dimensione

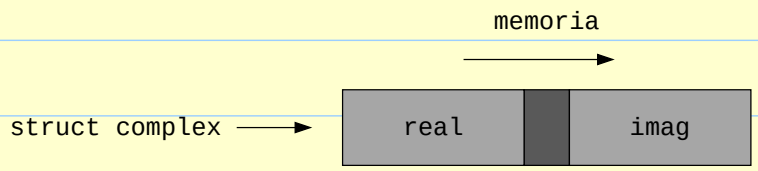
in memoria di una unione è quindi pari alla memoria necessaria per contenere il campo più grande, più eventuali *holes* di allineamento, ed è in genere molto più piccola della dimensione di una struttura.

Siccome non vi è modo di sapere quale è il campo attivo, è responsabilità del programmatore dare la corretta interpretazione dei valori contenuti nell'unione.

Per illustrare la differenza tra una struttura ed una unione consideriamo il tipo struttura:

```
struct complex {
        double real;
        double imag;
};
```

In una struttura i campi occupano zone di memoria separate e sequenziali, per cui la struttura `struct complex` appare in memoria come:

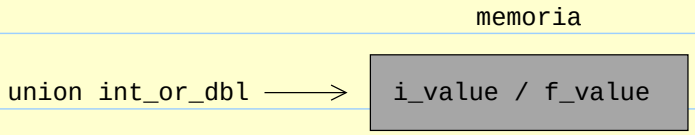


La zona più scura rappresenta eventuali *holes*.

Al tipo unione viene allocata una zona di memoria condivisa da tutti i suoi campi ma sufficiente a contenere il valore di un solo campo alla volta. Quindi, ad esempio, il tipo unione

```
union int_or_dbl {
        int    i_value;
        double f_value;
};
```

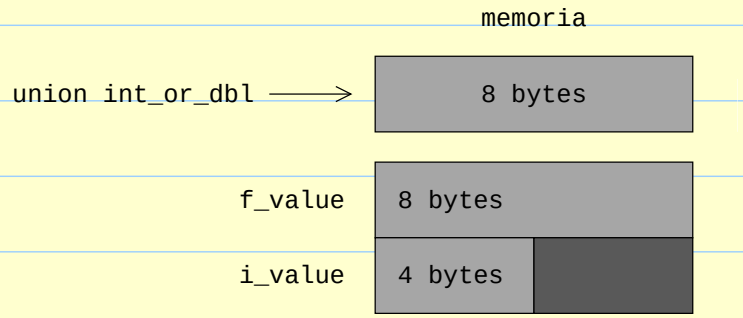
appare in memoria come:



I campi `i_value` e `f_value` condividono la stessa zona di memoria, quindi l'unione può contenere il valore di `i_value` o di `f_value`, ma non di entrambi contemporaneamente.

Possiamo pensare al tipo struttura come ad una scatola divisa in scompartimenti separati, ciascuno del quale ha un suo nome. Il tipo unione, invece, è come una scatola senza scompartimenti nella quale si possono mettere cose diverse, anche se una sola alla volta.

Se i campi di una unione hanno dimensione diversa, come nel caso dell'esempio, il valore del campo viene *sempre* memorizzato a partire dall'indirizzo di memoria dove inizia l'unione:



per cui l'indirizzo di `int_or_dbl.i_value` è sempre uguale a quello di `int_or_dbl.f_value` ed entrambi sono uguali a quello dell'unione. La zona più scura indica la zona di memoria assegnata all'unione ma non utilizzata quando viene memorizzato il valore del campo di dimensione più piccola. In contenuto di questa zona dipende da quello che era contenuto nell'unione prima dell'assegnazione del valore al campo di dimensione più piccola.

Data la differente organizzazione in memoria, in una struttura tutti i campi sono *attivi* contemporaneamente e non interferiscono tra di loro per cui il cambiamento di un campo non influisce sul valore degli altri. In una unione al contrario tutti i campi condividono la stessa, o parte della stessa, zona di memoria e quindi può essere *attivo* un solo campo alla volta. L'assegnazione di un valore ad un campo automaticamente distrugge qualsiasi valore fosse contenuto nell'unione.

Nel precedente esempio se viene assegnato un valore al campo `i_value`, una successiva assegnazione di un valore al campo `f_value` cancella il valore precedente di `i_value`, come mostra il seguente programma.

Esempio: `test-union.c`

```c
#include <stdio.h>

union int_or_flt {
        int   i_value;
        float f_value;
} number;

int main(void)
{
  int   i;
  float f;

  number.f_value = 5.0;      /* unione contiene float      */

  i = number.i_value;        /* Dipende dal sistema        */
  f = number.f_value;        /* Unione contiene float-> OK */

  printf("i: %10d \t f: %f\n", i, f);

  number.i_value = 3;        /* unione contiene int, float perso  */
```

```
  i = number.i_value;       /* Unione contiene int -> OK  */
  f = number.f_value;       /* Dipende dal sistema        */

  printf("i: %10d \t f: %f\n", i, f);

  number.i_value = 2.5;     /* unione contiene int */

  i = number.i_value;       /* Unione contiene int -> OK  */
  f = number.f_value;       /* Dipende dal sistema        */

  printf("i: %10d \t f: %f\n", i, f);

  return 0;
}
```

L'output di questo programma dipende dalla rappresentazione in memoria dei dati utilizzata dal sistema, sul mio ha dato:

```
i: 1084227584     f: 5.000000
i:          3     f: 0.000000
i:          2     f: 0.000000
```

Infatti quando assegnamo il valore `5.0` al campo `f_value` questo viene memorizzato nella zona di memoria associata all'unione `union number` nella rappresentazione utilizzata dal sistema per i numeri floating-point di tipo `float`. Se il valore viene riletto utilizzando il campo `i_value` il valore dei *bits* viene interpretato come la rappresentazione di un numero intero di tipo `int` e quindi il valore di `i_value` dipenderà dalla rappresentazione utilizzata dal sistema per i tipi `float` e `int`.

Osserviamo che sfruttando questo comportamento è possibile avere informazioni sulla rappresentazione dei dati utilizzata dal sistema, come mostra il seguente programma che stampa in formato esadecimale la rappresentazione di un numero floating-point di tipo `float`.

Esempio: `dirty-union.c`

```
#include <stdio.h>

union int_or_flt {
  int   i_value;    /* Si assume che il tipo int e float */
  float f_value;    /* abbiano la STESSA lunghezza       */
} number;

int main(void)
{
  number.f_value = 5.0;

  /* stampa il contenuto della memoria in formato esadecimale */
  printf("The machine representation of %e is %#010x\n",
         number.f_value, number.i_value);
  return 0;
}
```

Chiaramente l'utilizzo delle unioni in questo modo non è portabile in quanto sistemi diversi possono produrre risultati diversi a seconda della rappresentazione utilizzata. Sul mio ha prodotto

```
The machine representation of 5.000000e+00 is 0x40a00000
```

Le unioni sono spesso usate nelle comunicazioni poichè i dati trasferiti lungo un canale di trasmissione sono sempre di un solo tipo. Il tipo però dipende dal tipo operazione che si deve fare, ad esempio se è una ricezione o una trasmissione. Il seguente esempio illustra l'uso delle unioni in una semplice simulazione di connessione e trasferimento dati ad un nodo remoto.

Esempio: `simple_connection.c`

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

const int OPEN  =  0;
const int SEND  =  1;
const int CLOSE =  2;

struct message {
  int msg_id;
  union {
    int  data;
    char msg[41];
  } msg_body;
};

void remote(struct message pkt);

int main(void)
{
  struct message pkt;

  /* Apre la connessione */
  pkt.msg_id = OPEN;
  strcpy(pkt.msg_body.msg,"Hello from me");
  remote(pkt);

  /* Manda i dati */
  pkt.msg_id = SEND;
  pkt.msg_body.data = 10;
  remote(pkt);

  /* Chiude  la connessione */
  pkt.msg_id = CLOSE;
  strcpy(pkt.msg_body.msg,"Bye Bye");
  remote(pkt);

  return 0;
}
```

```
/* ----
 * remote()
 *
 * nodo remoto
 */
void remote(struct message pkt)
{

  if (pkt.msg_id == OPEN) {
    printf("Connection established....\n");
    printf("you said: %s\n\n", pkt.msg_body.msg);
    return;
  }

  if (pkt.msg_id == SEND) {
    printf("Data received....\n");
    printf("you sent: %d\n\n", pkt.msg_body.data);
    return;
  }

  if (pkt.msg_id == CLOSE) {
    printf("Closing connection....\n");
    printf("you said: %s\n\n", pkt.msg_body.msg);
    return;
  }

  printf("Connection failed.\n");
  return;
}
```

Quando il programma viene eseguito si ha

```
Connection established....
you said: Hello from me

Data received....
you sent: 10

Closing connection....
you said: Bye Bye
```

In questo semplice esempio la struttura `struct message` definisce il pacchetto inviato al nodo remoto. Il primo campo della struttura è un identificatore per il tipo di operazione da effettuare: `OPEN`, `SEND`, `CLOSE`. Il secondo campo contiene i dati. Siccome questi possono dipendere da tipo di operazione, ma una volta specificata l'operazione sono univocamente determinati, si usa una unione. Quale dei campi dell'unione è quello attivo viene determinato dal tipo di operazione da effettuare, come mostrato nella funzione `remote()`.

In generale i dati possono essere più complessi di un semplice dato di tipo `int`, inoltre il canale di comunicazione può essere usato sia in trasferimento che in ricezione, per cui una struttura più completa per un pacchetto potrebbe essere:

```
const int OPEN    = 0;
const int SEND    = 1;
const int RECEIVE = 2;
const int CLOSE   = 3;

struct message {
  int msg_id;
  union {
    struct open_ch     open_data;
    struct send_ch     send_data;
    struct receive_ch  write_data;
    struct close_ch    close_data;
  } msg_data;
};
```

in cui il tipo dati scambiati nelle quattro operazioni possibili sono specificati da quattro strutture.

2.43. Puntatori, Funzioni, Strutture ed Unioni (Rev. 2.5.1)

2.43.1. Puntatori

I puntatori al tipo struttura ed unione seguono le regole usuali dei puntatori per cui vengono definiti utilizzando l'operatore di referenza "`&`" e dereferenza "`*`". Ad esempio le istruzioni seguenti:

```
struct student {
     char first_name[61];      /* Nome      */
     char  last_name[61];      /* Cognome   */
     char     std_id[11];      /* Matricola */
     int           grade;      /* Voto      */
};

struct student *studente_corso_p;
struct student  studente_1;

studente_corso_p = &studente_1;
```

definiscono la variabile `studente_corso_p` di tipo puntatore a `struct student` e gli assegnano l'indirizzo di memoria di una struttura di tipo `struct student`.

Per accedere ai campi di una struttura attraverso un puntatore al tipo struttura si usa l'operatore di selezione "`->`", ottenuto premendo successivamente il carattere "`-`" ed il carattere "`>`". La sintassi è simile a quella dell'operatore di selezione "`.`":

```
pointer->field-name
```

dove `pointer` è un'espressione di tipo "puntatore a tipo struttura" o "puntatore a tipo unione" e `field-name` il nome del campo del tipo corrispondente. Così ed esempio

```
studente_corso_p->grade
```

fornisce il valore del campo `grade` della struttura `studente_1`.

È possibile accedere ai campi di una struttura attraverso un puntatore al tipo struttura anche utilizzando l'operatore di dereferenza "`*`" e l'operatore di selezione "`.`" con la sintassi

```
(*pointer).field_name
```

L'espressione `pointer->field` è per definizione esattamente equivalente a `(*pointer).field` e quindi si può usare si una sintassi che l'altra.

Osserviano che le parentesi "`()`" sono necessarie poichè l'operatore di selezione "`.`" ha precedenza più alta dell'operatore di dereferenza "`*`" ed associatività a sinistra per cui

```
*pointer.field_name
```

verrebbe interpretata come

```
*(pointer.field_name)
```

con il risultato di considerare il valore del campo della struttura come un indirizzo di memoria e di dereferenziarne il contenuto. In realtà questo viene segnalato come errore dal compilatore perchè l'operatore di selezione "`.`" può essere utilizzato solo con strutture o unioni e non con puntatori a strutture o unioni.

Nello Standard C il puntatore `pointer` può essere il puntatore nullo. In questo caso, se si applica l'operatore di indirizzo `&`, si ottiene l'*offset* in *bytes* del campo nella struttura. Ad esempio il seguente programma

Esempio: `structure-offset.c`

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

int main(void)
{
  struct student {
    char first_name[61];     /* Nome      */
    char  last_name[61];     /* Cognome   */
    char     std_id[11];     /* Matricola */
    int           grade;     /* Voto      */
  };

  int off;

  off = (int) (char *) &((struct student *) NULL)->first_name;
  printf("off: %d\n", off);

  off = (int) (char *) &((struct student *) NULL)->last_name;
  printf("off: %d\n", off);

  off = (int) (char *) &((struct student *) NULL)->std_id;
  printf("off: %d\n", off);

  off = (int) (char *) &((struct student *) NULL)->grade;
```

```
    printf("off: %d\n", off);

    return 0;
}
```

scrive l'offset dei campi della struttura `struct student`. Quando il programma viene eseguito sul mio computer produce il seguente output:

```
off: 0
off: 61
off: 122
off: 136
```

che mostra chiaramente che tra il campo `std_id` ed il campo `grade` c'è un *hole* di `3` *bytes*.

Lo Standard C fornisce la *macro* `offsetof(type,field)` che restituisce l'offset in bytes del campo di una struttura. Questa è definita nel file di header di sistema `stddef.h` usualmente come

```
#define offsetof(TYPE, FIELD) ((size_t) &((TYPE *)0)->FIELD)
```

per cui nell'esempio precedente per conoscere l'offset del campo `grade` avremmo potuto utilizzare

```
off = offsetof(struct student, grade);
```

includendo ovviamente il file `stddef.h`.

Ogni qual volta è possibile applicare l'operatore di referenza "`&`" ad una struttura è in generale possibile applicarlo anche ad un suo campo ottenendo così un "puntatore ad un campo di una struttura" che permette di accedere direttamente al campo all'interno della struttura senza dover specificare la struttura stessa. L'eccezione a questa regola è quando il campo è un *bit field*. Il motivo è che un campo di tipo *bit field* ha una lunghezza in bits fissata e quindi non è detto che inizi all'inizio di una locazione di memoria. Ad esempio se una locazione di memoria è composta da 8 bits questa può contenere due campi di tipo bit field di 4 bits ciascuna per cui chiaramente non è possibile associare un indirizzo ai due campi.

2.43.2. Funzioni e Strutture

Nel linguaggio C i parametri delle funzioni sono passati per valore per cui quando una struttura viene passata come parametro a una funzione ne viene effettuata una copia con scopo locale assegnando ai campi il valore dei campi della struttura passata. Se uno più campi della struttura sono arrays questi vengono copiati per intero. Un processo analogo si ha quando una funzione restituisce una struttura.

È chiaro che se una struttura contiene parecchi campi o campi di grosse dimensioni, come ad esempio arrays di grosse dimensioni, il passaggio per valore sia in ingresso che in uscita richiede un notevole dispendio di memoria e tempo ed è quindi poco efficiente. Per molte applicazioni è preferibile di conseguenza utilizzare funzioni che prendono come parametri e/o restituiscono come valore puntatori a strutture.

Il seguente programma mostra un esempio dell'uso delle funzioni con le strutture.

Esempio: `add_complex.c`

```
#include <stdio.h>

typedef struct {
        double re;
        double im;
} complex;

complex add_1(complex a, complex b);
complex add_2(complex *a, complex *b);
complex add_3(complex *a, complex *b);
void    add_4(complex *a, complex *b, complex *c);

int main(void)
{
  complex a, b, c;
  a.re = 1.0;
  a.im = 0.0;
  b.re = 0.0;
  b.im = 1.0;

  c = add_1(a, b);
  printf("add_1 says: (%f,%f)\n", c.re, c.im);

  c = add_2(&a, &b);
  printf("add_2 says: (%f,%f)\n", c.re, c.im);

  c = add_3(&a, &b);
  printf("add_3 says: (%f,%f)\n", c.re, c.im);

  add_4(&a, &b, &c);
  printf("add_4 says: (%f,%f)\n", c.re, c.im);

  return 0;
}


complex add_1(complex a, complex b)
{
  complex c;

  /* a, b e c tipo complex */
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}

complex add_2(complex *a_p, complex *b_p)
{
```

```
  complex c;

  /* a_p e b_p puntatori complex,  c tipo complex */
  c.re = (*a_p).re + (*b_p).re;
  c.im = (*a_p).im + (*b_p).im;
  return c;
}

complex add_3(complex *a_p, complex *b_p)
{
  complex c;

  /* a_p e b_p puntatori complex,  c tipo complex */
  c.re = a_p->re + b_p->re;
  c.im = a_p->im + b_p->im;
  return c;
}

void add_4(complex *a_p, complex *b_p, complex *c_p)
{
  /* a_p, b_p e c_p puntatori complex */
  c_p->re = a_p->re + b_p->re;
  c_p->im = a_p->im + b_p->im;
  return;
}
```

La funzione `add_1()` prende come parametri e ritorna *oggetti* di tipo `complex`, nella fattispecie come parametri i due numeri complessi da sommare e restituisce il valore della somma. In questo caso il valore dei campi delle strutture viene copiato sia in ingresso che in uscita.

Le funzioni `add_2()` ed `add_3()` differiscono solo nella sintassi utilizzata per accedere agli elementi della struttura attraverso il puntatore, mentre per il resto sono identiche. Entrambe prendono come parametri due puntatori ad oggetti di tipo `complex` e ritornano un oggetto di tipo `complex`. In questo caso il valore della struttura viene copiato solo in uscita, per cui le funzioni sono più efficienti della funzione `add_1()`.

Il passaggio dei valori può essere eliminato del tutto, come mostra la funzione `add_4()` che utilizza sia in ingresso che in uscita tre puntatori ad oggetti di tipo `complex` e non ritorna nulla. Questa funzione è la più efficiente delle quattro, anche se un pò risultare scomoda da usare.

2.44. Array di Strutture od Unioni (Rev. 2.0.2)

Strutture ed arrays possono essere combinate insieme. Non solo i campi delle strutture possono essere arrays ma è possibile definire arrays i cui elementi siano strutture. Supponiamo ad esempio di aver definito un tipo struttura per contenere le informazioni sul voto finale degli studenti di un corso:

```
struct student {
```

```
    char first_name[61];      /* Nome      */
    char  last_name[61];      /* Cognome   */
    char     std_id[11];      /* Matricola */
    int           grade;      /* Voto      */
};
```

Possiamo allora dichiarare un array per contenere tutti gli studenti di un corso come:

```
#define MAX_STUDENTS 100

struct student my_students[MAX_STUDENTS];
```

La variabile `my_students` è un array di tipo `struct student`. Questo vuol dire che ogni elemento dell'array

```
my_students[0], my_students[1], ..., my_students[MAX_STUDENTS-1]
```

è un oggetto di tipo `struct student`. Di conseguenza la seguente porzione di programma

```
for (i = 0; i < MAX_FRIENDS; ++i) {
    printf("Nome: %s \t %d\n",
           my_students[i].last_name,
           my_students[i].grade);
}
```

stampa il cognome ed il voto di ciascun studente.

Sebbene l'operatore di indice "`[]`" e l'operatore di selezione "`.`" abbiano la stessa precedenza non è necessario utilizzare parentesi poichè la loro associatività a sinistra per cui l'espressione

```
my_students[i].last_name
```

viene correttamente interpretata come

```
(my_students[i]).last_name
```

ossia "il campo `last_name` dell'i-esimo elemento dell'array di strutture `my_students`".

Come tutti gli arrays anche gli arrays di strutture possono essere dichiarati dinamicamente per cui, ad esempio, per definire l'array `my_students` avremmo potuto ugualmente usare le istruzioni

```
int max_students;
struct student *my_students;    /* puntatore a struct student */

max_students = 100;
my_students = (struct student *)
                  malloc(max_students * sizeof(struct student));
```

Come ulteriore esempio la seguente porzione di programma definisce dinamicamente un'array di numeri complessi.

```
typedef struct {
    double re;  /* real part of a complex number      */
    double im;  /* imaginary part of a complex number */
```

```
} complex;

int size;
complex *z;      /* puntatore ad un oggetto complex */

size = 100;
z = (complex *) malloc(size * sizeof(complex));
```

La parte reale ed immaginaria di ciascun numero dell'array è data rispettivamente da `z[i].re` e `z[i].im` mentre `z[i]` rappresenta l'i-esimo numero complesso. Per cui se ad esempio volessimo stamparli sullo schermo dovremmo usare istruzioni del tipo

```
for (i = 0; i < size; ++i) {
  printf("Re(z) = %f \t Im(z) = %f\n", z[i].re, z[i].im);
}
```

Naturalmente spesso non vi è un solo modo di risolvere un problema per cui per trattare una sequenza di numeri complessi avremmo anche potuto definire una struttura i cui campi sono arrays contenenti la parte reale ed immaginaria dei numeri complessi:

```
#define N 100

typedef struct {
  double re[N];
  double im[N];
} complex_arr;

complex_arr z_arr;
```

In questo caso la parte reale dell'i-esimo numero complesso è data da `z_arr.re[i]` mentre la pare immaginaria da `z_arr.im[i]` per cui le istruzioni di stampa diventano

```
for (i = 0; i < size; ++i) {
  printf("Re(z) = %f \t Im(z) = %f\n", z_arr.re[i], z_arr.im[i]);
}
```

Di nuovo le parentesi non sono necessarie poichè, come già osservato precedentemente, sebbene gli operatori "`[]` e "`.`" abbiano la stessa precedenza la loro associatività e a sinistra per cui l'espressione

```
z_arr.re[i]
```

viene correttamente interpretata come

```
(z_arr.re)[i]
```

ossia "l'i-esimo elemento dell'array campo della struttura".

2.45. Tipo Enumerativo (Rev. 2.0.2)

A volte può capitare che alcune variabili possano assumere solo un insieme finito di valori, l'esempio tipico che si fa in questi casi è quello di una variabile che contenga i giorni della

settimana. Dal momento che l'insieme di valori che la variabile può assumere è finito è sempre possibile associare arbitrariamente a ciascun valore possibile un numero intero e rappresentare la variabile con una variabile di tipo `int`. Ad esempio nel caso dei giorni della settimana potremmo associare il valore 0 alla domenica, il valore 1 al lunedì e così via fino al valore 6 per il sabato cosicchè i giorni della settimana possono essere rappresentati dalla variabile

```
int giorno;
```

che prende solo i valori $\{0, 1, 2, 3, 4, 5, 6\}$.

Sebbene non vi sia nulla di sbagliato in questa procedura è chiaro che utilizzare direttamente i valori numerici nella programmazione può non solo essere piuttosto scomodo ma anche generare programmi poco trasparenti, con tutte le conseguenza del caso. Ad esempio per sapere a che giorno della settimana si riferisce il controllo

```
if (giorno == 3) ....
```

è ovviamente necessario risalire alla corrispondenza esplicita utilizzata tra i numeri interi ed i giorni della settimana.

Per ovviare a questi inconvenienti è quindi preferibile utilizzare al posto dei valori numerici degli *identificatori* o *tag*, associati ai valori numerici, in modo da rendere la programmazione più chiara. Vi sono più modi introdurre i *tags*, ad esempio si potrebbe utilizzare il preprocessore C per cui nel caso dei giorni della settimana potremmo definire le macros

```
#define SUNDAY     0
#define MONDAY     1
#define TUESDAY    2
#define WEDNESDAY  3
#define THURSDAY   4
#define FRIDAY     5
#define SATURDAY   6
```

cosicchè il precedente controllo diventerebbe

```
if (giorno == WEDNESDAY) ....
```

molto più chiaro della precedente scrittura.

Il preprocessore C non riconosce le istruzioni del linguaggio C per cui non controlla che la sintassi delle istruzioni sia corretta. Questo è spesso causa di errori difficili da scoprire, soprattutto se la definizione delle macros è "piuttosto lontana" dal punto in cui si è prodotto l'errore. Se non si vuole quindi utilizzare macros si può adottare una soluzione equivalente definendo delle costanti intere, ad esempio per i giorni della settimana

```
const int sunday   = 0;
const int monday   = 1;
const int tuesday  = 2;
const int wednsday = 3;
const int thursday = 4;
const int friday   = 5;
const int saturday = 6;
```

Questa soluzione ha il vantaggio che utilizza solo istruzioni del linguaggio C, con annesso controllo di sintassi, ma al pari della definizione delle macros può risultare "scomoda" da scrivere. Per semplificare la definizione ed utilizzo di costanti intere da associare ai possibili valori di una variabile il linguaggio C fornisce il tipo *enumerativo*.

Il tipo enumerativo è formato da un insieme di valori interi identificati da identificatori o *tags* che vengono specificati all'atto della definizione del tipo. Gli identificatori, chiamati anche costanti enumerative, sono costanti di tipo `int`. Il tipo enumerativo viene definito utilizzando l'istruzione `enum`

```
enum enum-tag {tag_1, tag_2, ..., tag_n} enum-var;
```

dove `enum-tag` è l'identificatore o nome del tipo enumerativo, `tag_1`, ..., `tag_n` le costanti enumerative (*tags*) e `enum-var` il nome della variabile enumerativa di tipo `enum-tag` che può prendere i valori associati alle costanti enumerative `tag_1`, ..., `tag_n`.

La sintassi della dichiarazione del tipo enumerativo delle variabili di tipo enumerativo segue regole simili a quella delle strutture e delle unioni per cui per dichiarare la variabile di tipo enumerativo `giorno` si può usare sia l'istruzione

```
enum {sunday, monday, tuesday, wednsday, thursday,
      friday, saturday} giorno;
```

che le istruzioni

```
enum days {sunday, monday, tuesday, wednsday, thursday,
      friday, saturday};
enum days giorno;
```

Entrambe queste scritture dichiarano la variabile enumerativa `giorno` La differenza tra la prima scrittura e la seconda è che quest'ultima definisce anche il tipo enumerativo `enum days` che può quindi essere utilizzato ad esempio per dichiarare altre variabili.

Il tipo enumerativo viene rappresentato internamente come un tipo intero che però può dipendere dal sistema. In ogni caso, tuttavia, il tipo enumerativo è compatibile con i tipi interi e viene trattato come tale. Questo vuol dire i tipi enumerativi possono essere utilizzati in ogni contesto in cui è lecito utilizzare espressioni di tipo intero.

Il tipo enumerativo viene realizzato associando ad ciascuna costante enumerativa un valore intero secondo le seguenti regole:

- È possibile assegnare esplicitamente un valore alle costanti enumerative scrivendo

  ```
  tag = expression
  ```

  nelle definizione di tipo enumerativo, dove `expression` è un'espressione costante di tipo intero che può includere tags il cui valore sia stato già assegnato;

- Se non viene assegnato esplicitamente un valore, alla prima costante enumerativa viene assegnato il valore 0.

- Se non viene assegnato esplicitamente un valore, alle costanti enumerative seguenti viene assegnato il valore uguale alla costante enumerativa che la precede aumentato di 1.

Ad esempio nella definizione

```
enum days {sunday,
           monday,
           tuesday = 8,
           wednsday,
           thursday = tuesday + 3,
           friday,
           saturday}
```

alle costanti enumerative verranno assegnati i valori interi

```
sunday    ⇒  0
monday    ⇒  1
tuesday   ⇒  8
wednsday  ⇒  9
thursday  ⇒  11
friday    ⇒  12
saturday  ⇒  13
```

Non solo è possibile associare alle costanti enumerative qualsiasi valore di tipo `int` con segno, ma è anche possibile associare lo stesso valore a costanti enumerative diverse

```
enum color {red = 0, green = -2, blue = -2};
```

Nonostante la somiglianza con le strutture e le unioni, nello Standard C il tipo enumerativo è trattato semplicemente come un modo comodo di definire di costanti di tipo `int`, o poco più. Una conseguenza di ciò è che mentre il nome del tipo enumerativo `enum-tag` ha la stessa classe di memorizzazione e scopo dei nomi delle strutture, gli identificatori `tag` delle costanti enumerative sono memorizzate nella stessa classe di memorizzazione dei nomi delle variabili, delle funzioni e dei tipi dichiarati con `typedef` ed hanno lo stesso scopo di una variabile fosse definita nello stesso posto nel programma. Questo vuol dire ad esempio che mentre nella dichiarazione

```
enum color {red, green, blue} color;
```

gli l'identificatore `color` è associato sia al tipo enumerativo `enum color` che alla variabile di tipo enumerativa di tipo `enum color`, nelle istruzioni seguenti

```
int red;
enum color {red, green, blue} var;
```

la dichiarazione del tipo enumerativo `enum color` con una costante enumerativa di nome `red` nasconde la dichiarazione precedente `int red` dell'identificatore `red`.

Lo standard C non richiede che vengano fatti controlli sui valori assegnati alle variabili enumerative, di conseguenza essendo il tipo enumerativo rappresentato come un tipo intero è possibile assegnare ad variabile di tipo enumerativa un valore costante di tipo intero non necessariamente associato ad una costante enumerativa.

```
enum color {red = 0, green = 1, blue = 2} var;
var = 3;
```

Alcuni compilatori, tuttavia, in questi casi producono un messaggio di *warning*.

Come nota di stile si consiglia tuttavia di trattare i tipi enumerativi come se fossero differenti dai tipi interi ed quindi non usarli in espressioni con interi senza cast espliciti. Questo rende inoltre il programma più portabile poichè lo Standard C lascia libertà ai compilatori di effettuare controlli di compatibilità sui tipi enumerativi.

2.46. Operatore `sizeof` (Rev. 2.0.1)

L'operatore `sizeof` è un operatore `unario`, ossia che prende un solo operando. Il suo valore è di tipo costante intera senza segno e pari alla dimensione dell'operando.[9] Lo Standard C richiede che il valore di `sizeof` sia di tipo `size_t` (tipo *size*) definito nel file di sistema `stddef.h`.

L'operando può essere di due tipi. Nella prima forma l'operando è il nome o identificatore di un tipo racchiuso tra le parentesi "`()`"

```
sizeof(type)
```

In questo caso il valore di `sizeof` è la dimensione del tipo `type`, ossia il numero di unità di memoria occupate da un oggetto qualsiasi di tipo `type`. Ad esempio

```
sizeof(int)
```

fornisce la dimensione di un qualsiasi oggetto di tipo `int`. Nel computo della memoria occupata da un oggetto viene incluso qualsiasi *hole* necessario per il suo corretto l'allineamento in memoria, ad esempio nel caso di strutture od unioni, per cui l'operatore `sizeof` fornisce il numero di unità di memoria effettivamente occupate dall'oggetto.

Nella seconda forma l'operando è un'espressione, eventualmente racchiusa tra parentesi "`()`".

```
sizeof expression
sizeof (expression)
```

In questo caso il risultato è come se si fosse applicato l'operatore `sizeof` al tipo dell'espressione `expression`. Ad esempio

```
int i, s;
s = sizeof i;
```

è equivalente a

```
s = sizeof(int)
```

L'associatività dell'operatore `sizeof` è a destra ed il suo livello di precedenza è piuttosto alto. Di conseguenza

```
sizeof i*2
```

viene interpretato come

```
(sizeof i)*2
```

[9] Nello Standard C l'operando non può essere un campo di una struttura od unione di tipo bit field.

piuttosto che come

```
sizeof(i*2)
```

per cui l'uso delle parentesi può risultare necessario.

L'operatore `sizeof` non effettua nessuna conversione sull'espressione prima di determinarne la dimensione, così ad esempio può essere utilizzato per determinare la dimensione totale di un array senza che il nome dell'array venga convertita in puntatore. Ad esempio

```
int a[10];
s = sizeof a;
```

è equivalente a

```
s = sizeof(int) * 10;
```

e non a

```
s = sizeof(int *);
```

Osserviamo che

```
s = sizeof(a[10]);
```

non fornisce la dimensione totale dell'array ma quella di un suo elemento. Infatti la precedente istruzione è equivalente a

```
s = sizeof(int);
```

Tuttavia se l'espressione contiene operatori che effettuano conversioni queste sono effettuate per determinare il tipo del valore dell'espressione e quindi la dimensione stessa dell'espressione. Così ad esempio

```
int i;
sizeof(i*2.0)
```

è equivalente a

```
sizeof(double)
```

perchè il risultato dell'operazione è di tipo `double`. Osserviamo che le parentesi sono necessarie altrimenti

```
sizeof i*2.0
```

sarebbe interpretato come

```
sizeof(int)*2.0
```

Anche se le conversioni all'interno di un'espressione sono effettuate l'espressione non è valutata. La dimensione di un'espressione è infatti data dal tipo dell'espressione determinato in fase di compilazione analizzando l'espressione ed effettuando, se necessario, le dovute conversioni al solo scopo di determinarne il tipo del valore, e non dal suo valore determinato invece in fase di esecuzione. Ad esempio nelle seguenti istruzioni

```
int i;
i = 10;
printf("%d ->> %d\n", i, sizeof i++);
printf("%d ->> %d\n", i, sizeof ++i);
printf("%d ->> %d\n", i, sizeof (i++));
```

il valore della variabile `i` non è incrementato poichè la dimensione delle espressioni è valutata in fase di compilazione determinandone il tipo del loro valore, per cui le precedenti istruzioni sono a tutti gli effetti equivalenti a

```
printf("%d ->> %d\n", i, sizeof(int));
printf("%d ->> %d\n", i, sizeof(int));
printf("%d ->> %d\n", i, sizeof(int));
```

Infine l'operatore `sizeof` non può avere come operatore un tipo incompleto, perchè non potrebbe determinarne la dimensione, a meno che questo non sia un parametro formale di una funzione. In questo caso infatti il parametro viene convertito a puntatore al tipo ed il valore restituito dall'operatore `sizeof` è la dimensione del puntatore al tipo. Ad esempio

```
int s;
extern int a[];
s = sizeof(a);              /* Errata */
```

è errata perchè l'array `a` è definita da qualche altra parte e non è quindi possibile stabilirne la dimensione. Al contrario

```
void f(int a[])
{
  int s;
  s = sizeof(a);      /* Corretto */
  return;
}
```

è corretta perchè l'istruzione

```
s = sizeof(a);
```

viene interpretata come

```
s = sizeof(int *);
```

Questo comportamento non dovrebbe sorprendere più di tanto in quanto il parametro formale `a` è a tutti gli effetti un puntatore al tipo `int` la cui dimensione è ben determinata.

2.47. Operatori bit-a-bit (Rev. 2.0.2)

La più piccola quantità di informazione memorizzabile nella memoria di un computer è il *bit*. Il *bit* può assumere sono due valori di solito indicati con `1` e `0` anche se, a seconda del loro utilizzo, si anche usano altre rappresentazioni come `true/false` o `yes/no`. Tutti i tipi discussi fino ad ora sono rappresentati nella memoria di un computer come una sequenza di più bits, ad esempio il tipo `char` utilizza una unità di otto bits, ossia un *byte*.

A volte, tuttavia, può risultare utile poter accedere ai singoli bit poichè la manipolazione dei bits permette una programmazione di livello più basso, e quindi più potente, della programmazione che utilizza solo tipi. Per contro un programma di basso livello *bit-oriented* spesso risulta molto meno trasparente di un programma di più altro livello *type-oriented*.

La programmazione *bit-oriented* si usa ad esempio per programmi che usano particolari codifiche, come i programmi di compressione, trasmissione ed immagazzinamento dei dati. In fisica la programmazione *bit-oriented* viene usata spesso quando si utilizzano variabili che possono assumere solo pochi valori, ad esempio due (*spin*), per cui è possibile ridurre la richiesta di memoria utilizzando i singoli bit, o gruppi di pochi bits, per memorizzare le variabili (*multi-spin coding*).

2.47.1. Rappresentazione dei dati binari

Nella programmazione *bit-oriented* i dati sono sequenze di bits (*dati binari*), e quindi di difficile lettura per un umano. È quindi necessario utilizzare qualche tipo di codifica che li renda più facilmente intellegibili. La codifica più comoda per rappresentare i dati binari è il formato *esadecimale* (*hex*) poichè ogni *hex-digit* fornisce il valore di quattro bits. Ad esempio in questo modo il contenuto di un byte è dato da soli due hex-digit.

La seguente tavola riporta la conversione tra la codifica esadecimale e quella binaria.

Tavola conversione hex - binaria

hex	Binario		hex	Binario
0	0000		8	1000
1	0001		9	1001
2	0010		A	1010
3	0011		B	1011
4	0100		C	1100
5	0101		D	1101
6	0110		E	1110
7	0111		F	1111

Per le lettere si possono usare anche i caratteri minuscoli: `a`, `b`, `c`, `d`, `e` ed `f`.

Ad esempio se un byte contiene i dati binari "`11001000`" il suo contenuto nella rappresentazione hex viene ottenuto utilizzando la tavola di conversione dopo aver raggruppato i bits in gruppi da quattro a partire da desta, per cui nel caso specifico si ha `0xC8`. Il prefisso "`0x`", o anche "`0X`", è necessario per indicare che si tratta di un hex-digit. Siccome per gli hex-digits si possono usare indifferentemente lettere maiuscole e minuscole avremmo potuto ugualmente scrivere `0xc8`.

Se avessimo usato la rappresentazione *decimale* il contenuto del byte sarebbe stato rappresentato da `200`, di interpretazione in termini di bits molto meno immediata di quella esadecimale. Infatti per passare da un numero hex alla rappresentazione *binaria* è sufficiente utilizzare la tavola di conversione associando a ciascun hex-digit la configurazione dei quattro bits corrispondenti. Nel caso della rappresentazione decimale invece la conversione richiede divisioni

successive per 2. Il vantaggio della rappresentazione esadecimale rispetto a quella decimale per i dati binari è quindi evidente.

Operatori bit-a-bit

Per operare sui singoli bits dei dati binari il linguaggio C fornisce sei operatori detti *operatori bit-a-bit* che possono essere raggruppati in tre categorie:

- Operatori logici bit-a-bit;
- Operatore di complemento bit-a-bit;
- Operatori di shift (*traslazione*);

I dati binari sono semplici sequenze di bits senza nessuna particolare codifica di conseguenza gli operatori bit-a-bit prendono come operandi solo espressioni a valore di tipo intero `int` o `char`. Inoltre poichè ai bits dei dati binari non è associata nessuna particolare codifica questi operatori trattano ogni bit degli operandi individualmente ed indipendentemente gli uni dagli altri, da cui il nome di operatori bit-a-bit.

Sebbene gli operandi degli operatori bit-a-bit possono essere sia con che senza segno per una maggiore sicurezza si consiglia di utilizzare gli operatori bit-a-bit solo con tipi interi senza segno. Infatti il comportamento di questi operatori con i tipi interi con segno può differire da un computer ad un altro soprattutto se il computer non utilizza la rappresentazione in complemento a due per i numeri interi negativi.

2.47.2. Operatori logici bit-a-bit

I operatori logici bit-a-bit sono in ordine di precedenza:

`&`	⇒	*and*
`^`	⇒	*or esclusivo* o *xor*
`|`	⇒	*or inclusivo* o semplicemente *or*

La loro associatività è a *sinistra*.

Gli operatori logici bit-a-bit sono operatori binari, ossia prendono due operandi. Il risultato dell'operatore è un dato binario dello stesso tipo degli operandi in cui ciascun bit è il risultato della funzione booleana applicata ai due bits corrispondenti per posizione nei due operandi. Se gli operandi sono di tipo diverso si applicano le usuali regole di conversione tra tipi ed il risultato sarà del tipo in cui gli operandi sono stati convertiti.

L'azione degli operatori logici bit-a-bit su ciascun singolo bit è:

- L'operatore *and* "`&`" restituisce un bit `1` se e solo se entrambi i bits su cui opera sono bit `1`, altrimenti restituisce un bit `0`

- L'operatore *or esclusivo* "^ " restituisce un bit 1 solo nel caso in cui solo uno dei due bits su cui opera è un bit 1, altrimenti restituisce un bit 0.
- L'operatore *or inclusivo* "|" restituisce un bit 1 se almeno uno dei due bits su cui opera è un bit 1, altrimenti restituisce un bit 0.

La loro azione può essere riassunta dalla seguente "*tavola della verità*":

Operatori Logici Bit-a-Bit

bit 1	*bit 2*	*bit 1 & bit2*	*bit 1 ^ bit2*	*bit 1 | bit2*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Gli operatori logici bit-a-bit sono commutativi ed associativi per cui il compilatore è libero di riarrangiare le espressioni contenenti questi operatori. Di conseguenza le espressioni con operatori logici bit-a-bit non sono valutate necessariamente nell'ordine con cui sono scritte ma a seconda del compilatore in una delle possibili forme equivalenti.

A prima vista gli operatori logici bit-a-bit sembrano molto simili agli operatori logici "&&" (*and*) e "||" (*or*). Ad esempio se entrambi gli operandi dell'operatore logico "&&" sono `true` ("non zero") il risultato sarà `true`, allo stesso modo in cui sarà 1 il risultato dell'operatore logico bit-a-bit "&" se entrambi i bits su cui opera sono 1. Tuttavia nonostante molte similitudini gli operatori logici e gli operatori logici bit-a-bit sono operatori diversi. Infatti mentre gli operatori logici operano sul valore degli operandi come un tutt'uno, gli operatori logici bit-a-bit operano su ciascun bit degli operandi indipendentemente.

Per illustrare questa differenza supponiamo di voler controllare se i due numeri interi `i` e `j` sono entrambi non nulli. Questo si ottiene facilmente ad esempio come:

```
if ( (i != 0) && (j != 0) ) {
   printf("Sono entrambi diversi da zero\n");
}
```

Alternativamente, siccome un'espressione è considerata da un punto di vista logico `true` se il suo valore è non nullo, il controllo può essere effettuato anche come

```
if ( i && j ) {
   printf("Sono entrambi diversi da zero\n");
}
```

Se in questa seconda versione avessimo usato l'operatore logico bit-a-bit "&" al posto dell'operatore logico "&&" e scritto

```
if ( i & j ) {
   printf("Sono entrambi diversi da zero\n");
}
```

il controllo sarebbe risultato errato. Infatti se `i = 1` e `j = 2` il test fallisce e il programma non scrive "`Sono entrambi diversi da zero`", anche che i numeri sono chiaramente non nulli.

Il motivo è che l'operatore logico bit-a-bit opera su ogni bit degli operandi separatamente e non sul valore degli operandi per cui il risultato dell'operazione `i & j` è:

```
i = 1    00000001
j = 2    00000010
         --------
i & j    00000000
```

ed il test giustamente fallisce.

Osserviamo che se l'operatore logico bit-a-bit "`&`" fosse stato usato nella prima versione del test

```
if ( (i != 0) & (j != 0) ) {
    printf("Sono entrambi diversi da zero\n");
}
```

invece si avrebbe avuto la risposta corretta. Infatti il risultato dell'espressione logica `(i != 0)` è il valore booleano `0` o `1` a seconda che l'affermazione sia vera o falsa. Per cui, sebbene l'uso dell'operatore logico bit-a-bit "`&`" sia improprio in questo contesto, il risultato è nonostante tutto corretto poichè il valore degli operandi dell'operatore bit-a-bit sono sempre convertiti nei valori interi `0` e `1` che differiscono sono per il primo bit. Da questo esempio possiamo concludere che il risultato degli operatori logici bit-a-bit è lo stesso degli operatori logici solo se gli operandi sono espressioni a valore booleano `0` o `1`.

Vi è infine un'altra differenza tra gli operatori logici bit-a-bit e gli operatori logici che a volte può causare errori piuttosto difficili da trovare. Infatti mentre gli operatori logici non valutano l'operando a destra se il valore dell'operando a sinistra è sufficiente per determinare se il valore dell'espressione logica è *true* o *false*, gli operatori logici bit-a-bit valutano sempre entrambi gli operandi.

2.47.3. Operatore di complemento bit-a-bit

L'operatore di complemento bit-a-bit o negazione bit-a-bit:

| `~` ⇒ *not bit-a-bit* o *bit flip* |

è un operatore unario che ritorna l'inverso o negazione o complemento bit-a-bit del suo operando:

- L'operatore "`~`" restituisce un bit `1` se il suo operando è un bit `0` ed un bit `0` se il suo operando è un bit `1`.

La sua tavola della verità è quindi semplicemente

Complemento

bit	~*bit*
0	1
1	0

L'operatore di complemento bit-a-bit è simile all'operatore logico "`!`" (*not*), ma anche in questo caso la somiglianza è solo apparente poichè l'operatore bit-a-bit non opera sul valore dell'operando ma indipendentemente su ogni bit della sua rappresentazione binaria. Di conseguenza ogni bit della rappresentazione binaria di di `~e` è l'inverso di quello che era nell'operando `e`. Ad esempio se `e` è un tipo intero a 8-bit allora

$$\begin{aligned} \texttt{e} &= \texttt{0xc8} \quad \texttt{11001000} \\ \texttt{~e} &= \texttt{0x37} \quad \texttt{00110111} \\ \texttt{e} &= \texttt{0xF0} \quad \texttt{11110000} \\ \texttt{~e} &= \texttt{0x0F} \quad \texttt{00001111} \end{aligned}$$

Se l'operando dell'operatore unario "`~`" non è di tipo intero, questo viene convertito automaticamente ad un tipo intero prima di applicargli l'operatore.

Se l'operando `e` dell'operatore di complemento bit-a-bit è di intero con segno il valore di `~e` dipende dalla rappresentazione utilizzata per i numeri interi con segno.

La rappresentazione più utilizzata per gli interi con segno è quella di *complemento a due*. Ricordiamo che la rappresentazione in complemento a due con n bits del numero intero negativo $-i$ si ottiene aggiungendo `1` al complemento bit-a-bit della rappresentazione binaria con $n-1$ bits del numero intero positivo i. In questa rappresentazione il bit più significativo, ossia quello più a sinistra, vale `0` per i numeri positivi e `1` per quelli negativi. Per convincersi di ciò basta osservare che se ad una stringa di bits si somma il suo complemento bit-a-bit si ottiene una stringa di bits tutti uguali ad `1`. Se ora a questa stringa si somma `1` si otterrà una stringa di bits tutti uguali a `0` ossia la rappresentazione del valore `0`. Di conseguenza la stringa con tutti i bits uguali ad `1` è la rappresentazione in complemento a due del numero intero negativo -1 e quindi il valore della stringa più il suo complemento bit-a-bit è -1 in complemento a due.

Esempio:

Valore i	**Binario**	**~i**	**−i**	**Valore −i**
72	01001000	10110111	10111000	−72
9	00001001	11110110	11110111	−9
1	00000001	11111110	11111111	−1
0	00000000	11111111	00000000	0
−5	11111011	00000100	00000101	5

Non tutti i computers usano la rappresentazione in complemento a due per gli interi con segno, di conseguenza il valore del risultato dell'operatore di complemento bit-a-bit applicato ad interi con segno può differire da un computer all'altro. Per questo si consiglia di utilizzare l'operatore di complemento bit-a-bit solo con operandi di tipo intero senza segno.

Per un operando `e` di tipo intero senza segno il valore di `~e` è `UINT_MAX - e` se il valore di `e` è, o è convertito ad, `unsigned int` e `ULONG_MAX - e` se invece il valore di `e` è, o è convertito ad, `unsigned long int`. Nello Standard C i valori di `UINT_MAX` e `ULONG_MAX` sono definiti nel file di header di sistema `limits.h`.

2.47.4. Operatori di shift

Vi sono due operatori binari di *shift* (*traslazione*):

`<<`	⇒	shift a sinistra *left-shift*
`>>`	⇒	shift a destra *right-shift*

Entrambi gli operatori hanno associatività a sinistra ed hanno la stesso livello di precedenza.

Ciascun operatore prende due operandi entrambi di tipo intero e nel caso servisse le usuali regole di conversione sono applicate separatamente ad entrambi gli operandi. Il risultato degli operatori di shift è del tipo, o del tipo in cui è convertito, l'operando di sinistra.

Il primo operando degli operatori di shift è l'oggetto i cui bits vanno traslati mentre il secondo operando è il numero di posizioni di cui vanno traslati i bits del primo operando. Il valore dell'operatore è uguale al valore del primo operando dopo lo shift. La direzione di traslazione dipende dall'operatore usato. L'operatore left-shift "`<<`" sposta a sinistra tutti i bits del primo operando del numero di posizioni specificato dal secondo operando. I bits più a sinistra (*most-significative*) che vengono spostati fuori sono persi, mentre i bits nelle posizioni più a destra (*less-significative*) che si liberano sono messi a `0`. Analogamente, l'operatore right-shift "`>>`" sposta a destra tutti i bits del primo operando del numero di posizioni specificato dal secondo operando. I bits spostati fuori a destra sono persi, mentre il valore assegnato ai bits che si liberano a sinistra dipende dal compilatore. Lo Standard C richiede che l'operazione di right-shift sia o *logica* o *aritmetica*. Nel primo caso, shift logico, ai bits che si liberano a sinistra viene assegnato il valore di bit `0` mentre nel secondo caso, shift aritmetico, a questi viene assegnato il il valore di bit `0` se l'operando è senza segno o il valore del bit più significativo prima dello shift se l'operando è con segno. Questo vuol dire che il right-shift aritmetico assegna ai bits che si liberano a sinistra il valore del bit del segno per variabili con segno, e quindi il valore di bit `0` per valori positivi e `1` per valori negativi, ed il valore di bit `0` per le variabili senza segno. Per lo più i compilatori usano un right-shift aritmetico, tuttavia questa duplice possibilità fa si che l'uso dell'operatore "`>>`" con variabili con segno produca programmi generalmente non portabili.

Il valore degli operatori di shift è indeterminato se il valore del secondo operando è negativo, per cui uno shift a sinistra di un numero negativo di posizioni non necessariamente risulta in uno shift a destra, e viceversa. Il risultato può risultare indeterminato anche se il valore del secondo operando è più grande od uguale alla dimensione in bit del primo operando. Se invece il valore del secondo operando è `0` non viene effettuato nessuno shift.

Esempio:

`signed char c`	`c = 0x2b`	00101011	43
`c << 1`	`c = 0x56`	01010110	86
`c >> 1`	`c = 0x15`	00010101[a]	21
`c >> 2`	`c = 0x0a`	00001010[a]	10
`c >> 3`	`c = 0x0a`	00000101[a]	5
`(c << 2) >> 2`	`c = 0xeb`	11101011[b]	−21
	`c = 0x2b`	00101011[c]	43

[a] shift logico o aritmetico
[b] shift aritmetico
[c] shift logico

Spostare a sinistra di una posizione equivale a moltiplicare per 2, di due posizioni a moltiplicare per 4, e così via. Quindi un shift a sinistra di n posizioni è equivalente a moltiplicare per 2^n. Se l'operatore left-shift equivale a moltiplicare per 2, l'operatore right-shift equivale alla divisione tra interi per 2, per cui uno shift a destra di n posizioni è equivalente a dividere per 2^n. Siccome le operazioni di shift sono più veloci di quelle di divisione e moltiplicazione il compilatore sostituisce quando possibile tutte le divisioni e moltiplicazioni per 2^n con gli *shifts* equivalenti.

Come per gli operatori di assegnamento anche per gli operatori bit-a-bit vale la forma contratta

```
var op= expr  <==>  var = var op expr
```

Ad esempio

```
i &= j;          /* equivale a i = i & j;  */
i <<= 3;         /* equivale a i = i << 3; */
```

Esercizi

1. Utilizzando gli operatori bit-a-bit scrivere un programma che stampa i bits di un numero intero sia in formato binario che in formato esadecimale.

2.48. Esempio: Semplice algoritmo di criptaggio (Rev. 2.0.2)

Un modo molto semplice di *criptare* una stringa di caratteri è quello di sostituirne ogni carattere, o gruppo di caratteri, con un altro secondo una regola data. Chiaramente affinchè la regola sia utilizzabile deve essere invertibile, ossia deve essere possibile decriptare la stringa.

Esistono diverse regole di criptaggio più o meno sofisticate, nella seguente, molto semplice, il criptaggio viene effettuato invertendo il valore di alcuni bits della rappresentazione binaria di ciascun carattere della stringa scelti con l'ausilio di una "chiave" (*key*) di criptaggio che fornisce la posizione dei bits da invertire. La chiave può essere rappresentata come una sequenza di bit `0` e bit `1` con i bits `0` nelle posizioni dei bits della rappresentazione binaria del carattere da lasciare inalterati e bits `1` in quelle dei bits da invertire. Se ogni carattere

è rappresentato da 8 bits esistono $2^8 - 1 = 255$ chiavi diverse di criptaggio, escludendo ovviamente quella composta da tutti 0 (*identità*). Poichè se il valore di un bit viene invertito due volte il suo valore resta inalterato il decriptaggio della stringa si effettua utilizzando la stessa regola di criptaggio, come mostra il seguente esempio:

Criptaggio:

```
A        =   01000001   65 (ASCII)
key      =   00000111
             ----------
A ^ key  =   01000110   70 (ASCII)  ⇒  F
```

Decriptaggio

```
F        =   01000110   70 (ASCII)
key      =   00000111
             ----------
F ^ key  =   01000001   65 (ASCII)  ⇒  A
```

Per poter realizzare questo tipo di criptaggio si deve accedere ai singoli bit della rappresentazione binaria del carattere per cui è necessario l'utilizzo degli operatori bit-a-bit ed in particolare, in questo caso, dell'operatore logico bit-a-bit *xor* "^" (*or esclusivo*) come mostra il seguente programma.

Programma: `cript_string.c`

```
/*****************************************************************

- Descrizione : Cripta una stringa con chiave di 8-bits data.

- Moduli      : bit_utils.c

- $Id: cript_string.c v 1.2 08.11.04 AC
*****************************************************************/
#include <stdio.h>
#include <string.h>
#include "bit_utils.h"

int cript_string(char *str, unsigned char key);

int main(void)
{
  unsigned short int key;   /* Chiave di criptaggio */
  char          line[100];  /* Stringa da criptare  */

  printf("stringa: ");
  fgets(line, sizeof(line),stdin);

  printf("key    : ");
  scanf("%hu", &key);
```

```
  /* solo 256 differenti chiavi con 8 bit
     inclusa l'identita'                    */
  key %= 256;

  printf("key    : %d => ", key);
  print_bit((unsigned char *) &key, sizeof(char));

  cript_string(line, key);
  printf("\nStringa Criptata:\n%s", line);

  cript_string(line, key);
  printf("\nStringa Decriptata:\n%s", line);

  return 0;
}

int cript_string(char *str, unsigned char key)
{
  int c;

  for (c = 0; c < strlen(str) - 1; ++c) {
    str[c] ^= key;
  }
  return c;
}
```

Il criptaggio/decriptaggio della stringa viene effettuato dalla funzione `cript_string()` che prende come parametri la stringa da criptare e la chiave di criptaggio e ritorna il numero di caratteri criptati.

Il programma prende come input la stringa da criptare/decriptare ed un numero che fornisce la chiave di criptaggio. Con 8 bit vi sono solo 256 chiavi differenti inclusa l'identità

```
key %= 256;
```

Per controllare il programma si può criptare/decriptare la stringa `"A"` con la chiave 7:

```
stringa: A
key    : 7
key    : 7 => 00000111

Stringa Criptata:
F

Stringa Decriptata:
A
```

Osserviamo che non tutti i 256 caratteri possibili sono "stampabili" sullo schermo per cui può capitare che il risultato del programma sia piuttosto indecifrabile, ossia *criptato*!

2.49. Esempio: Stampa bit-a-bit e Modulo `bit_utils.c` (Rev. 2.0.3)

Quando si utilizzano direttamente i bits in una programmazione bit-oriented è utile avere a disposizione delle funzioni che permettano di stampare un dato binario sia in formato binario che esadecimale. Queste possono essere scritte facilmente utilizzando gli operatori bit-a-bit come mostra la seguente funzione stampa il contenuto di un intero di tipo `int`.

Esempio: `print_byte_int.c`

```
/*****************************************************************

- Descrizione : stampa un intero byte per byte in formato
                esadecimale.
                Assume ordinamento little-endian ed
                il tipo char di 8 bits (due hex-digit).

- $Id: print_byte_int.c v 1.2 02.11.03 AC
*****************************************************************/
#include <stdio.h>

void print_byte_int(int n)
{
  unsigned int    n_char;                /* # char in int */
  unsigned int     n_bit;                /* # bit  in int */
  unsigned int byte_mask;
  int                chr;

  n_char    = sizeof(int);
  n_bit     = 8 * n_char;
  byte_mask = 0xFF << (n_bit - 8);   /* mask = 0xFF000000 */

  for (chr = 0; chr < n_char; ++chr) {
    fprintf(stdout, "%02x ",
            (unsigned int) (n & byte_mask) >> (n_bit - 8));
    n <<= 8;
  }
  fputc('\n', stdout);

  return;
}
```

Questa funzione assume che il tipo `char` sia di 8 bits e che l'ordinamento dei bytes nel tipo `int` sia little-endian con i bytes ordinati da destra a sinistra.

Affinchè i bytes siano stampati in ordine con il più significativo a sinistra ed il meno significativo a destra i bytes di `n` vanno letti in ordine a partire dal più significativo che in questo caso è quello più a sinistra. Per estrarre il contenuto del primo byte a sinistra si utilizza una "maschera" `byte_mask` i cui bits del primo byte a sinistra sono tutti bit `1` ed i restanti bit `0`. Questa è costruita traslando a sinistra di `n_bit - 8` posizioni il byte 11111111 =`0xFF`.

```
byte_mask = 0xFF << (n_bit - 8);
```

Il passo successivo è quello di estrarre il byte più a sinistra del numero intero `n`. Questo si ottiene facilmente con un and logico bit-a-bit

```
n & byte_mask
```

Infatti la rappresentazione binaria del risultato di questa operazione contiene nel primo byte a sinistra i bits del primo byte a sinistra di `n` mentre tutti gli altri bits sono bit `0`. Il risultato viene infine traslato a destra di `n_bit - 8` posizioni in modo da poterne stampare il `valore` in formato esadecimale con due hex-digit "`%02x`". Se la traslazione a destra è un right-shift aritmetico questa potrebbe inserire dei bit `1` non desiderati, di conseguenza prima di effettuare la traslazione il risultato viene convertito ad `unsigned int` con in cast esplicito.

```
(unsigned int) (n & byte_mask) >> (n_bit - 8)
```

Una volta stampato il primo byte l'intero `n` viene traslato a sinistra di `8` posizioni

```
n <<= 8;
```

in modo che adesso nel primo byte a sinistra si trova il secondo byte da sinistra di `n` che può quindi essere stampato con le stesse operazioni appena descritte. Questa operazione viene ripetuta per tutti i bytes di `n`

```
for (chr = 0; chr < n_chr; ++chr) {
    fprintf(stdout, "%02x␣",
            (unsigned int) (n & byte_mask) >> (n_bit - 8));
    n <<= 8;
}
```

Per maggiore chiarezza i bytes sono stampati separando gli uni dagli altri con uno spazio bianco.

Naturalmente questa non è l'unica soluzione, nè la più flessibile, per stampare i bytes di un oggetto, ad esempio si potrebbe utilizzare un puntatore per scorrerne i bytes, come nella funzione `print_byte()` nel modulo seguente, invece di traslarlo ogni volta. Questa funzione è stata discussa con il solo scopo di illustrare l'uso degli operatori bit-a-bit.

Modulo

Il seguente modulo `bit_utils.c` fornisce due funzioni per stampare il contenuto di un oggetto generico sia in formato binario che byte a byte in formato esadecimale.

Header: `bit_utils.h`

```
/*************************************************************

- Definizioni per il modulo bit_utils.c

- Funzioni :

    void print_bit       (unsigned char *ptr, unsigned int size);
    void print_byte      (unsigned char *ptr, unsigned int size);
```

```
- Descrizione :

    print_bit()       -> stampa i bits di un oggetto
    print_byte()      -> stampa i bytes di un oggetto in hex

    ptr  ->  puntatore ad un oggetto
    size ->  dimenzione dell'oggetto (sizeof)

- Note :

    print_byte() assume che il tipo char sia di 8 bits

    print_bit() e print_byte() funzionano con
    l'ordinamento little-endian o big-endian.

- $Id: bit_utils.h v 2.1 31.10.04 AC
*************************************************************/
extern void print_bit       (unsigned char *ptr, unsigned int size);
extern void print_byte      (unsigned char *ptr, unsigned int size);
```

Le funzioni `print_bit()` e `print_byte()` stampano il contenuto di un oggetto rispettivamente in formato binario ed in formato esadecimale. Il primo argomento `ptr` è il puntatore all'oggetto, convertito in puntatore ad `unsigned char`, mentre il secondo argomento `size` è la dimensione dell'oggetto come misurata dall'operatore `sizeof`.

La funzione `print_byte()` assume che il tipo `char` sia di 8 bits e quindi rappresentabile con due hex-digit, in caso contrario scrive sullo `stdout` un messaggio di avvertimento.

Per quanto riguarda l'ordinamento dei bytes le funzioni `print_bit()` e `print_byte()` assumono che l'ordinamento sia little-endian o big-endian, in caso contrario scrivono sullo `stdout` un messaggio di errore.

Modulo: `bit_utils.c`

```
/*************************************************************

- Descrizione : modulo bit_utils.c

- $Id: bit_utils.c v 2.1 31.10.04 AC
*************************************************************/
#include <stdio.h>
#include <limits.h>

/* L'ordinamento dei bytes dipende dall'architettura utilizzata
 */
#if __BYTE_ORDER == __LITTLE_ENDIAN          /* 1234 */

#define FIRST_CHAR(ptr, size) ptr + size - 1
#define NEXT_CHAR(ptr)        --(ptr)

#elif __BYTE_ORDER == __BIG_ENDIAN           /* 4321 */
```

```
#define FIRST_CHAR(ptr, size) ptr
#define NEXT_CHAR(ptr)        ++(ptr)

#endif    /* __BYTE_ORDER */

/* Funzioni Private */
static void print_bit_char(unsigned char c);
static void print_byte_char(unsigned char c);

/* ----
 * print_bit()
 *
 */
extern void print_bit(unsigned char *ptr, unsigned int size)
{
  unsigned char   *c;    /* puntatore a char di *ptr */
  int             chr;   /* char in *ptr             */

#if __BYTE_ORDER != __LITTLE_ENDIAN && __BYTE_ORDER != __BIG_ENDIAN
  printf("Byte Ordering not supported! \n\n");
  return;
#endif

  c = FIRST_CHAR(ptr, size);
  for (chr = 1; chr <= size; ++chr) {
    print_bit_char(*c);
    fputc(' ', stdout);
    NEXT_CHAR(c);
  }
  fputc('\n', stdout);

  return;
}

/* ----
 * print_byte()
 *
 */
extern void print_byte(unsigned char *ptr, unsigned int size)
{
  unsigned char *c;    /* puntatore a char di *ptr */
  int           chr;   /* char in *ptr             */

#if __BYTE_ORDER != __LITTLE_ENDIAN  &&  __BYTE_ORDER != __BIG_ENDIAN
  printf("Byte Ordering not supported! \n\n");
  return;
#endif

  c = FIRST_CHAR(ptr, size);
  for (chr = 1; chr <= size; ++chr) {
    print_byte_char(*c);
```

```
    fputc(' ', stdout);
    NEXT_CHAR(c);
  }
  fputc('\n', stdout);

  return;
}

/* ----
 * print_bit_char()
 *
 */
static void print_bit_char(unsigned char c)
{
  int           bit;          /* bit in char               */
  unsigned char mask;         /* maschera di CHAR_BIT bits */

  mask = 1 << (CHAR_BIT - 1);          /* mask = 10000....0 */
  for (bit = 1; bit <= CHAR_BIT; ++bit) {
    fputc( ((c & mask) == 0) ? '0' : '1', stdout);
    c <<= 1;
  }
  return;
}

/* ----
 * print_byte_char()
 *
 */
static void print_byte_char(unsigned char c)
{
  if (CHAR_BIT != 8) {
    fprintf(stdout, "Your char is not 8 bit long\n");
    return;
  }
  fprintf(stdout, "%02x", c);
  return;
}

#undef FIRST_CHAR
#undef NEXT_CHAR
```

Note sul modulo `bit_utils.c`

Il modulo utilizza le due funzioni private `print_bit_char()` e `print_byte_char()` per stampare il contenuto di un byte rispettivamente in formato binario o esadecimale.

- `#if __BYTE_ORDER == __LITTLE_ENDIAN  /* 1234 */`

```
#define FIRST_CHAR(ptr, size) ptr + size - 1
#define NEXT_CHAR(ptr)        --(ptr)

#elif __BYTE_ORDER == __BIG_ENDIAN          /* 4321 */

#define FIRST_CHAR(ptr, size) ptr
#define NEXT_CHAR(ptr)        ++(ptr)

#endif    /* __BYTE_ORDER */
```

L'ordinamento dei bytes in oggetti di dimensione superiore al byte dipende dall'architettura del computer. Per conoscere l'ordinamento utilizzato e stampare i bit nell'ordine corretto il modulo utilizza la macro `__BYTE_ORDER` definita nel file di sistema `endian.h`. Nel caso in cui l'ordinamento utilizzato sia little-endian con il byte più significativo più a destra la macro viene definita come `__LITTLE_ENDIAN`. Se invece l'ordinamento utilizzato è big-endian in cui byte più significativo è quello più a sinistra `__BYTE_ORDER` viene definita come `__BIG_ENDIAN`. Per trattare i due casi contemporaneamente vengono definite le macros `FIRST_CHAR` che fornisce il byte meno significativo dell'oggetto e `NEXT_CHAR` per passare da un byte al successivo.

- **Funzione** `print_bit_char()`

  - `mask = 1 << (CHAR_BIT - 1);      /* mask = 10000....0 */`

  Per stampare i bits ordinati in modo che il primo bit stampato sia il bit più a sinistra di `c` (*most-significative* bit) e l'ultimo quello più a destra (*less-significative* bit), si utilizza la "*maschera*" `mask` il cui most-significative bit è il bit `1` e gli altri sono bit `0`. La maschera si costruisce facilmente spostando il bit `1` a sinistra di `CHAR_BIT - 1` posizioni.

  Per aumentare la portabilità il modulo usa la macro `CHAR_BIT` definita nel file di sistema `limits.h` che fornisce il numero di bits usati dal computer per rappresentare il tipo `char`.

  - `fputc( ((c & mask) == 0) ? '0' : '1', stdout);`

  Per conoscere il valore del bit più significativo di `c` si effettua un and logico bit-a-bit con la maschera `mask` che ha tutti i bits uguali al bit `0` tranne il più significativo. Il valore del risultato di `c & mask` è diverso da zero se e solo se il bit più significativo di `c` è il bit `1` di conseguenza

  ```
  ((c & mask) == 0) ? '0' : '1'
  ```

  sarà il carattere '`0`' o '`1`' a seconda che il bit più significativo di `c` sia il bit `0` o il bit `1`. Per scrivere si utilizza la funzione

  ```
  int fputc(int c, FILE *stream);
  ```

  il cui prototipo è definito nel file di header `stdio.h`, che scrive il carattere `c` su `stream`. In questo caso quindi la funzione `fputc()` scriverà sullo `stdout` il carattere '`0`' o '`1`' a seconda che il bit più significativo di `c` sia `0` o `1`.

- `c <<= 1;`

  Si spostano a sinistra di un posto tutti i bits di `c` in modo che il prossimo test con `mask` dia il valore del bit a destra del bit più significativo di `c`. Osserviamo che in questo modo il valore di `c` viene cambiato. Se non si vuole che questo accada lo stesso risultato si può ottenere spostando a destra di un posto i bits di `mask`

  ```
  mask >>= 1;
  ```

  In questo caso è essenziale che `mask` sia un tipo senza segno, altrimenti nel caso di right-shift aritmetico il test sarebbe errato in quanto ai bits lasciati liberi a sinistra verrebbe assegnato il valore del bit più a sinistra di `mask`, ossia il bit `1`, cosicchè l'operazione `c & mask` selezionerebbe il valore di più bits di `c`.

- **Funzione** `print_byte_char()`

  - ```
    if (CHAR_BIT != 8) {
      fprintf(stdout, "Your char is not 8 bit long\n");
      return;
    }
    ```

    La funzione stampa un byte in formato esadecimale con due hex-digit e quindi richiede che il tipo `char` sia rappresentato da 8 bits.

  - `fprintf(stdout, "%02x", *c);`

    Se ogni locazione di memoria contiene 8 bits è possibile rappresentarne il valore come due hex-digit mediante la direttiva di conversione "`%02x`".

- **Funzione** `print_bit()`

  - ```
    #if __BYTE_ORDER != __LITTLE_ENDIAN && __BYTE_ORDER != __BIG_ENDIAN
      printf("Byte Ordering not supported! \n\n");
      return;
    #endif
    ```

    Se l'ordinamento dei bytes non è nè little-endian nè big-endian la funzione stampa un messaggio di errore.

  - `c = FIRST_CHAR(ptr, size);`

    il puntatore `c` punta al byte meno significativo dell'oggetto, che sarà quello più a destra o più a sinistra a seconda dell'ordinamento utilizzato.

  - ```
    for (chr = 1; chr <= size; ++chr) {
      print_bit_char(*c);
      fputc('␣', stdout);
      NEXT_CHAR(c);
    }
    ```
    ```

Si stampa il contenuto di tutti i bytes dell'oggetto in formato binario utilizzando la funzione privata `print_bit_char()`. Per semplificare la lettura si raggruppano i bits in gruppi di `CHAR_BIT` bits separati da uno spazio bianco.

• **Funzione** `print_byte()`

La funzione `print_byte()` differisce dalla funzione `print_bit()` solo per l'utilizzo della funzione privata `print_byte_char()` per stampare il contenuto dei bytes in formato esadecimale.

Test del modulo

Il seguente programma illustra l'uso del modulo `bit_utils.c`

Programma: `test-bits.c`

```
/*****************************************************************

- Descrizione : scrive un char ed un int in
                formato binario ed esadecimale
                (test per bit_utils.c)

- $Id: test-bits.c v 2.2 01.11.04 AC
*****************************************************************/
#include <stdio.h>
#include <limits.h>
/*
 * modulo bit_utils.c
 */
#include "bit_utils.h"

void print_byte_int(int n);

int main(void)
{
  int         n;
  char        c;
  char line[81];

  printf("Carattere: ");
  fgets(line, sizeof(line), stdin);
  sscanf(line,"%c", &c);

  printf("\nchar '%c'\n", c);
  printf("Size                    : %d bits\n",
         sizeof(char) * CHAR_BIT);
  printf("Rapp. Binaria           : ");
  print_bit( (unsigned char *) &c, sizeof(char));
  printf("Rapp. Byte (hex)        : ");
  print_byte( (unsigned char *) &c, sizeof(char));
```

```
    printf("\nNumero intero: ");
    fgets(line, sizeof(line), stdin);
    sscanf(line,"%d", &n);

    printf("\nint %d\n", n);
    printf("Size                    : %d bits\n",
           sizeof(int) * CHAR_BIT);
    printf("Rapp. Binaria           : ");
    print_bit( (unsigned char *) &n, sizeof(int));
    printf("Rapp. Byte (hex)        : ");
    print_byte( (unsigned char *) &n, sizeof(int));
    printf("Rapp. Byte int (hex)    : ");
    print_byte_int(n);


    printf("\n");
    return 0;
}
```

Osserviamo che per poter stampare il contenuto della variabile `n` di tipo `int` utilizzando la funzione `print_bit()` l'indirizzo di memoria di `n` è trasformato al tipo `unsigned char` mediante un cast esplicito. In questo modo la memoria occupata da `n` viene vista dalla funzione `print_bit()` come un array di `sizeof(int)` elementi di tipo `unsigned char`.

Il programma deve essere compilato come

```
$ cc test-bits.c bit_utils.c print_byte_int.c
```

o in forma equivalente.

Quando il programma viene eseguito si ottiene

```
Carattere: a

char 'a'
Size                    : 8 bits
Rapp. Binaria           : 01100001
Rapp. Byte (hex)        : 61

Numero intero: 1234567

int 1234567
Size                    : 32 bits
Rapp. Binaria           : 00000000 00010010 11010110 10000111
Rapp. Byte (hex)        : 00 12 d6 87
Rapp. Byte int (hex)    : 00 12 d6 87
```

2.50. Ordinamento dei bytes nei tipi multi-byte (Rev. 2.0.2)

La struttura naturale di indirizzamento della memoria utilizzata dal linguaggio C è quella in cui ogni singolo carattere (o byte) della memoria può essere indirizzato singolarmente. Questo vuol dire che ogni byte della memoria ha un suo indirizzo che può essere assegnato ad un puntatore. Computers che usano questo tipo di indirizzamento sono chiamati *byte-addressable* computers.

I tipi di dimensione più grande di un byte, detti anche tipi *multi-byte*, occupano blocchi di memoria composti da più caratteri contigui ed il loro indirizzo in memoria è di solito dato dall'indirizzo del *primo carattere* del blocco definito come carattere del blocco con l'indirizzo più basso. Tuttavia l'ordine con cui i diversi bytes di un tipo multi-byte sono disposti nei vari caratteri del blocco di memoria può dipendere dall'architettura del computer e quindi il contenuto del primo carattere del blocco in genere dipende dal computer.

Le due architetture più usate sono l'architettura *big-endian* o *left-to-right* e *little-endian* o *right-to-left*.

- Nell'architettura big-endian i caratteri del blocco sono occupati con il byte più significativo nel primo carattere del blocco (*big-end-first*). In questa architettura l'indirizzo del tipo multi-byte corrisponde all'indirizzo del suo byte più significativo.
- Nell'architettura little-endian i caratteri del blocco sono occupati con il byte meno significativo nel primo carattere del blocco (*little-end-first*). In questa architettura l'indirizzo del tipo multi-byte corrisponde all'indirizzo del suo byte meno significativo.

Siccome la memoria viene occupata a partire dalle locazioni con indirizzo minore nell'ordinamento big-endian il tipo multi-byte viene messo in memoria partendo dal suo byte più significativo e quindi da sinistra a destra. Invece nell'ordinamento little-endian il tipo multi-byte viene messo in memoria partendo dal suo byte meno significativo e quindi da destra a sinistra.

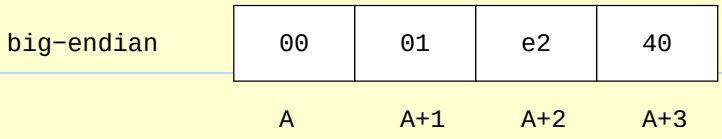
Per illustrare i due diversi ordinamenti consideriamo il numero intero 123456 memorizzato in un tipo intero di 32 bit, ovvero di 4 byte:

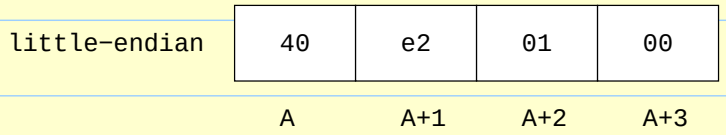
$$123456 = \texttt{00000000 00000001 11100010 01000000} = \texttt{00 01 e2 40}$$

A seconda dell'architettura i 4 bytes sono organizzati in memoria a partire dal byte più a sinistra o più a destra:

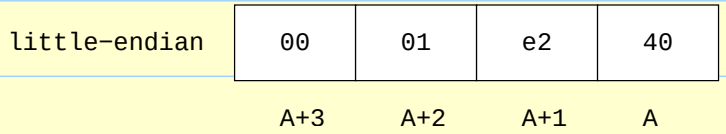
Indirizzo		**little-endian**	**big-endian**
A	⇒	01000000	00000000
A+1	⇒	11100010	00000001
A+2	⇒	00000001	11100010
A+3	⇒	00000000	01000000

Possiamo visualizzare i due ordinamenti come segue:





ovvero



da cui risulta evidente l'ordinamento da sinistra a destra o da destra a sinistra delle due architetture.

I seguente programma determina l'ordinamento usato dal computer scrivendo l'ordinamento in memoria di una variabile di tipo `int`.

Programma: `test-endian.c`

```
/*****************************************************************

- Descrizione : Determina l'ordinamento in memoria di un int
                Usa il modulo bit_utils.c

- $Id: test-endian.c v 1.2 01.11.04 AC
*****************************************************************/
#include <stdio.h>
/*
 * modulo bit_utils.c
 */
#include "bit_utils.h"

int main(void)
{
  int         n, add;
  unsigned char *c_p;
  char      line[81];

  printf("Numero intero: ");
  fgets(line, sizeof(line), stdin);
  sscanf(line,"%d", &n);

  printf("\nbit-rep: ");
  print_bit((unsigned char *) &n, sizeof(int));

  printf("\nbyte-rep: ");
  print_byte((unsigned char *) &n, sizeof(int));

  printf("\nmem-ord: \n");
  c_p = (unsigned char *) &n;
```

```
  for (add = 0; add < sizeof(int); ++add)
    {
      printf("%p : ", c_p + add);
      print_byte((unsigned char *) &c_p[add], sizeof(char));
    }

  return 0;
}
```

Per accedere ai singoli bytes della variabile di tipo `int n` si usa il puntatore `c_p` al tipo `unsigned char` a cui viene assegnato l'indirizzo della variabile `n` ossia l'indirizzo del primo carattere del blocco di memoria occupato dalla variabile `n`.

```
char *c_p;
...
c_p = (unsigned char *) &n;
```

In questo modo, siccome la dimensione di un `char` è 1, è possibile accedere ai singoli bytes di `n` semplicemente incrementando `c_p`. Il cast nell'assegnazione è necessario in quanto i puntatori sono a tipi diversi.

Chiaramente il risultato dell'ordinamento dipende dal computer, ad esempio sul mio ottengo:

```
Numero intero: 123456

bit-rep: 00000000 00000001 11100010 01000000

byte-rep: 00 01 e2 40

mem-ord:
0xbffffacc : 40
0xbffffacd : e2
0xbfffface : 01
0xbffffacf : 00
```

che mostra chiaramente l'ordinamento little-endian.

Vi sono molti modi di determinare l'ordinamento dei bytes utilizzato, ad esempio utilizzando una unione come mostra il programma seguente.

Programma: `test-endian_1.c`

```
/*****************************************************************

- Descrizione : Determina l'ordinamento dei bytes

- $Id: test-endian_1.c v 1.0 31.10.04 AC
*****************************************************************/
#include <stdio.h>

union {
  unsigned int                          n;
  unsigned char a[sizeof(unsigned int)];
```

```
} u;

int main(void)
{
  u.n = 1;

  if (u.a[0] == 1) {
    printf("Ordinamento little-endian\n");
  }
  else if (u.a[sizeof(unsigned int) - 1] == 1) {
    printf("Ordinamento big-endian\n");
  }
  else {
    printf("Ordinamento non riconosciuto\n");
  }

  return 0;
}
```

La filosofia del programma è la stessa del precedente. Il campo `a` dell'unione è un array di tipo `unsigned char` di dimensione pari alla dimensione di un tipo `int`. Ciascun elemento corrisponde ad un carattere del blocco in cui viene memorizzato il contenuto del campo `n` di tipo `int` dell'unione. Il campo `a` è l'equivalente di `A` nelle figure precedenti. Di conseguenza se assegnamo il valore `1` al campo `u.n` questo si troverà nell'elemento `a[0]` nel caso di ordinamento little-endian (little-end-first) e nell'elemento `a[sizeof(unsigned int) - 1]` nel caso di ordinamento big-endian (big-end-first). Se il valore non si trova in nessuno di questi due l'ordinamento non è nè big-endian nè little-endian.

I termini big-endian e little-endian derivano dai Lillipuziani dei Viaggi di Gulliver, il cui problema principale era se le uova bollite dovessero essere aperte dal lato grande (big-endian) o da quello piccolo (little-endian).

Il problema della conversione tra un ordinamento e l'altro è noto come il *NUXI problem*. Infatti se la parola UNIX fosse memorizzata in due variabili di 2 byte questa apparirebbe in memoria come UNIX in un sistema con architettura big-endian e NUXI in un sistema con architettura little-endian.

Osserviamo che l'ordinamento big-endian/little-endian si riscontra anche in altri campi, come ad esempio nel modo di scrivere le date. Gli Europei scrivono la data come `gg/mm/aa` quindi un ordinamento little-endian, mentre i Giapponesi la scrivono come `aa/mm/gg` e quindi usano un ordinamento big-endian. Per contro gli Americani la scrivono come `mm/gg/aa` e quindi ne big-endian ne little-endian, ma con ordinamento middle-endian. Architetture middle-endian con ordinamenti perversi dei bytes si possono trovare anche su alcuni computers, ad esempio i computers PDP.

A. Appendici

A.1. Compilazione ed esecuzione di un programma in C in ambiente UNIX (Rev. 2.1)

Sebbene la procedura generale per creare un file contenente un programma in linguaggio C, compilarlo ed eseguirlo sia la stessa su tutti i computers i dettagli specifici dipendono da tre elementi: sistema operativo, editore di testo e compilatore. Qui considereremo il caso di un computer con sistema operativo UNIX e compilatore GNU C Per gli altri sistemi rimandiamo alla documentazione allegata al computer.

Scrittura del Programma

Per scrivere un programma in linguaggio C, o in un qualsiasi altro linguaggio di programmazione, si utilizza un editore di testo, da non confondersi con un word-processor che genera documenti formattati, con cui si crea uno o più *files di testo* (files sorgente) contenenti la sequenza delle istruzioni in linguaggio C che formano il programma.

Il nome dei files non deve contenere spazi bianchi o altri caratteri speciali come "`/`" o "`*`" e deve terminare con il *suffisso* "`.c`" per indicare che il file contiene istruzioni in linguaggio C. Ad esempio si può usare "`myprogram.c`" o "`my-program.c`" ma non "`my program.c`" o "`my/program.c`".

Esistono molti editori di testo, uno molto comune su tutti i sistemi UNIX è l'editore GNU Emacs. Per editare un file in ambiente UNIX basta eseguire il comando:

```
$ emacs source.c
```

dove `source.c` è il nome del file che si vuole editare. Se il file `source.c` non esiste Emacs crea un nuovo file di nome `source.c`.

Compilazione

In ambiente UNIX il compilatore GNU C viene invocato con il comando:

```
$ cc source.c
```

ovvero

```
$ gcc source.c
```

dove `source.c` è il nome del file sorgente del programma in linguaggio C. Se il nome del file non finisce con il suffisso "`.c`" il sistema operativo potrebbe non riconoscere il file come file sorgente un programma in C generando di conseguenza un messaggio di errore. L'eseguibile prodotto dal compilatore si trova nel file chiamato "`a.out`".

Il comando `cc` in realtà non invoca solo il compilatore, ma chiama nell'ordine:

Preprocessore	:	che modifica la sorgente secondo le direttive contenute nella sorgente stessa
Compilatore	:	che traduce il programma generato dal preprocessore nel codice oggetto in linguaggio macchina
Linker	:	che crea l'eseguibile usando il codice oggetto prodotto dal compilatore e quello disponibile nelle librerie

Gli errori che si possono verificare a questo stadio sono errori di sintassi o di compilazione. In questo caso la compilazione viene interrotta ed il programma eseguibile non viene prodotto.

Il comando `cc` prende una serie di opzioni specificate mediante delle *flags*. Ad esempio se si desidera creare un file eseguibile con un nome diverso da `a.out` si può usare la flag "`-o`":

```
$ cc -o executable source.c
```

dove `executable` è il nome del file contenente l'eseguibile. Non è necessario che il file eseguibile termini con il suffisso "`.out`" e spesso l'eseguibile viene chiamato con lo stesso nome del file sorgente senza nessun suffisso.

Alternativamente è possibile cambiare il nome al file `a.out` utilizzando il comando UNIX "`mv`" (*move*):

```
$ cc source.c
$ mv a.out executable
```

Altre possibili flags sono "`-E`" per invocare solo il preprocessore, oppure "`-c`" per creare il codice oggetto senza chiamare il linker e quindi senza generare l'eseguibile. Nel primo caso il programma prodotto viene inviato sullo schermo, mentre nel secondo caso viene creato un file con lo stesso nome del file sorgente ma con il suffisso "`.o`". Altre flags molto utili in fase di sviluppo del programma sono

- "`-Wall`" per aumentare il livello di messaggi prodotti in fase di compilazione.
- "`-ansi`" o "`-std=c89`" o "`-std=iso9899:1990`" per selezionare il dialetto C89.
- "`-std=iso9899:199409`" per selezionare il dialetto C89 Amendment 1.
- "`-std=c99`" o "`-std=iso9899:1999`" per selezionare il dialetto C99. Al momento dello scrivere il compilatore GCC C supporta solo parzialmente lo Standard C (1999).
- "`-pedantic`" per segnalare come "warning" ogni disuniformità rispetto allo Standard C selezionato.
- "`-pedantic-errors`" per segnalare come "errore" ogni disuniformità rispetto allo Standard C selezionato.

Per una lista completa rimandiamo al manuale del compilatore consultabile con il comando "man":

```
$ man cc
```

ovvero

```
$ man gcc
```

o con il comando "info":

```
$ info gcc
```

Esecuzione

Un programma eseguibile viene eseguito semplicemente richiamando il suo nome, per cui se l'eseguibile si trova nel file `a.out` questo viene eseguito semplicemente con il comando:

```
$ a.out
```

ovvero se l'eseguibile si trova nel file `executable`

```
$ executable
```

Se il programma è invocto da una linea di comando quando questo termina sul terminale ritorna il *prompt* di sistema.

Eventuali errori durante l'esecuzione del programma sono chiamati errori di esecuzione e non necessariamente terminano l'esecuzione del programma. In genere questi errori possono risultare più difficili da trovare degli errori di compilazione.

Interruzione dell'esecuzione di un programma

A volte può essere necessario interrompere un programma in esecuzione, ad esempio perchè è entrato in un ciclo infinito. In ambiente UNIX questo si ottiene generalmente con premendo contemporaneamente la key "`CTRL`" e il carattere "`c`". In questo caso l'esecuzione viene interrotta e sullo schermo ritorna il prompt di sistema.

A.2. Breve introduzione a GNU Emacs (Rev. 2.0)

Emacs è un *editore* di testo disponibile su praticamente tutti i sistemi UNIX. Per invocare Emacs in un sistema UNIX basta eseguire il comando

```
$ emacs filename
```

dove `filename` è il nome del file che di vuole editare, ad esempio `my_c_program.c`. Se il file `filename` non esiste Emacs crea un nuovo file di nome `filename` nel direttorio in cui si stà lavorando altrimenti, se `filename` esiste, Emacs "apre" in file richiesto che può così essere

modificato. Infine se `filename` non viene dato Emacs apre una finestra vuota ("*buffer*") dove è possibile scrivere. In questo caso per non perdere quanto scritto bisogna *salvare* il contenuto del buffer in un file prima di uscire da Emacs.

Principali comandi di Emacs

I comandi di Emacs usano due keys speciali: la key `CONTROL` ("`CTRL`") e la key `META` di solito identificata con la key `ESCAPE` ("`ESC`"). Per indicare i comandi useremo la seguente notazione:

`C-<chr>`	⇒	premere contemporaneamente la key `CONTROL` ed il carattere "`chr`". Ad esempio `C-h` significa premere `CTRL` e `h` *contemporaneamente*.
`M-<chr>`	⇒	premere contemporaneamente la key `META` ed il carattere "`chr`". Ad esempio `M-h` significa premere `ESC` e `h` *contemporaneamente*.

Nel seguito sono riportati i comandi principali di Emacs. La documentazione completa può essere letta utilizzando il comando `info`:

```
$ info emacs
```

Terminare Emacs

`C-x C-c`	⇒	termina Emacs

Se il file è stato modificato Emacs chiede se si vogliono salvare le modifiche.

Files

`C-x C-f`	⇒	legge un file nel buffer
`C-x C-s`	⇒	salva il contenuto del buffer nel file che si stà modificando
`C-x i`	⇒	inserisce il contenuto di un file
`C-x C-w`	⇒	salva il contenuto del buffer su un file

In caso di errore

`C-g`	⇒	annulla il comando
`C-x u` o `C-_`	⇒	annulla l'ultima modifica

Emacs ricorda la sequenza di comandi fatti, per cui ripetendo più volte il comando `C-_` è possibile annullare le modifiche procedendo a ritroso.

Ricerca

C-s	⇒	cerca in avanti
C-r	⇒	cerca indietro
RET	⇒	termina la ricerca
C-g	⇒	annulla la ricerca

Per ripetere l'ultima ricerca in avanti o indietro usare C-s o C-r.

Movimento cursore

		Avanti	Indietro
Carattere	⇒	C-b o "→"	C-f o "←"
Parola	⇒	M-b	M-f
Linea	⇒	C-p o "↑"	C-n o "↓"
Fine Linea	⇒	C-e	
Inizio Linea	⇒	C-a	

Cancellare

		Avanti	Indietro
Carattere	⇒	DEL	C-d
Parola	⇒	M-DEL	M-d
Linea (fino alla fine)	⇒	M-0 C-k	C-k
Dal marcatore al cursore	⇒	C-w	
Inserisci ultima cosa cancellata	⇒	C-y	

Marcatore

C-@ o C-SPC ⇒ Inserisci marcatore alla posizione cursore

Sostituire

M-% ⇒ sostituisce una stringa di testo

Possibili azioni:

SPC	⇒	sostituisce e va alla prossima occorrenza
,	⇒	sostituisce senza spostare il cursore

`DEL` ⇒ non sostituisce e va alla prossima occorrenza
`|` ⇒ sostituisce tutte le occorrenze rimanenti
`RET` ⇒ termina ignorando le occorrenze rimanenti

Più finestre

`C-x 1` ⇒ chiude tutte le finestre meno quella dove é il cursore
`C-x 2` ⇒ divide la finestra in due
`C-x o` ⇒ sposta il cursore in un'altra finestra

Buffer

`C-x b` ⇒ selezione un altro buffer
`C-x C-b` ⇒ lista tutta i buffers
`C-x k` ⇒ chiudi un buffer

Help

`C-h` ⇒ help
`C-h t` ⇒ tutorial interattivo
`C-h a` ⇒ ricerca di un comando
`C-h c` ⇒ descrive l'azione di un comando

A.3. Miniguida ai comandi essenziali di UNIX (Rev. 2.0)

Riportiamo qui di seguito i comandi essenziali per utilizzare il sistema operativo UNIX.

Comando		Descrizione
`cd`	⇒	cambia la directory corrente
`cd ..`	⇒	cambia la directory corrente nella directory subito "sopra"
`cp`	⇒	copia un file su un altro
`exit`	⇒	termina la shell
`file`	⇒	determina il tipo file
`grep`	⇒	ricerca una stringa in un file
`less`	⇒	mostra il contenuto di un file sul terminale
`logout`	⇒	termina la sessione
`ls`	⇒	lista il contenuto di una directory
`ls -l`	⇒	lista estesa del contenuto di un a directory
`mkdir`	⇒	crea una directory

Comando		Descrizione
`more`	⇒	mostra il contenuto di un file sul terminale
`mv`	⇒	cambia nome ad un file o directory
`passwd`	⇒	cambia la propria password
`ps`	⇒	mostra i processi attivi
`pwd`	⇒	mostra il nome della directory corrente
`rm`	⇒	cancella un file
`rmdir`	⇒	cancella una directory
`reset`	⇒	reset del terminale
`whoami`	⇒	mostra il nome della login

Per ottenere maggiori informazioni su un comando si può usare il comando `man`, ad esempio `man ls` mostra i dettagli del comando `ls`. Per cercare i comandi relativi ad un argomento si può utilizzare il comando `apropos` che fornisce la lista di tutti comando connessi con un argomento. Ad esempio per conoscere i comandi relativi alle directories basta dare il comando `apropos directory`.

A.4. Sistemi di numerazione digitale (Rev. 2.0)

I sistemi di numerazione digitali rappresentano i numeri con una serie di simboli (*digit*) il cui valore dipende sia dal valore del simbolo che dalla sua posizione nella stringa, per questo sono chiamati *sistemi posizionali*. A seconda della scelta del numero di simboli usati si hanno differenti sistemi di numerazione.

I sistemi di numerazione utilizzati sui computers sono il sistema decimale, il sistema binario, il sistema ottale e quello esadecimale.

A.4.1. Sistema decimale

Il sistema decimale usa i 10 simboli:

0 1 2 3 4 5 6 7 8 9

per questo viene anche chiamato sistema in base 10. Il valore di un numero in notazione decimale è dato dalla somma del valore di ciascun digit moltiplicato per il valore posizionale 10^k con k intero non-negativo o negativo a seconda che il digit si trovi a sinistra o a destra del punto decimale. Il valore di k aumenta spostandosi dal punto decimale sia verso destra che verso sinistra. Ad esempio il valore di 2754.214 in notazione decimale è

$$2654.214_{10} = 2 \times 10^3 + 6 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 + 2 \times 10^{-1} + 1 \times 10^{-2} + 4 \times 10^{-3}$$

A.4.2. Sistema binario

Il sistema binario usa solo due simboli:

0 1

per cui è un sistema in base 2 e di conseguenza il valore posizionale di ciascun digit è 2^k dove k segue le stesse regole del sistema decimale:

$$1011.101_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 11.625_{10}$$

Conversione Decimale → Binario

Per trasformare un numero dall'usuale rappresentazione decimale a quella binaria basta dividere successivamente per 2 la sua parte intera scrivendo da destra a sinistra il resto della divisione, e moltiplicare successivamente per 2 la sua parte decimale scrivendo da sinistra a destra la parte intera del risultato. Come esempio mostriamo che $25.375_{10} = 11001.011_2$:

$$\begin{array}{rcll} 25/2 & = & 12 + \text{Resto } 1 & \Rightarrow \quad 1 \\ 12/2 & = & 6 + \text{Resto } 0 & \Rightarrow \quad 0 \\ 6/2 & = & 3 + \text{Resto } 0 & \Rightarrow \quad 0 \\ 3/2 & = & 1 + \text{Resto } 1 & \Rightarrow \quad 1 \\ 1/2 & = & 0 + \text{Resto } 1 & \Rightarrow \quad 1 \\ \hline & & & \Rightarrow 25_{10} = 11001_2 \end{array} \qquad \begin{array}{rcll} 0.375 \times 2 & = & 0.750 & \Rightarrow \quad 0 \\ 0.750 \times 2 & = & 1.500 & \Rightarrow \quad 1 \\ 0.500 \times 2 & = & 1.000 & \Rightarrow \quad 1 \\ & & & \\ & & & \\ \hline & & & \Rightarrow 0.375_{10} = 0.011_2 \end{array}$$

A.4.3. Sistema ottale

Il sistema ottale usa gli 8 simboli:

0 1 2 3 4 5 6 7

ed è quindi un sistema in base 8 con valore posizionale 8^k:

$$372.15_8 = 3 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 + 1 \times 8^{-1} + 5 \times 8^{-2} = 250.203125_{10}$$

L'utilità del sistema ottale risiede nella semplicità con cui è possibile passare dalla rappresentazione binaria a quella ottale e viceversa per cui il sistema ottale fornisce un modo più compatto di rappresentare dati in rappresentazione binaria.

Conversione Decimale → Ottale

Per trasformare un numero dall'usuale rappresentazione decimale a quella ottale si procede come per la trasformazione in rappresentazione binaria con la sola differenza che in questo caso si divide e moltiplica per 8 e non per 2.

Conversione Ottale ↔ Binario

La conversione tra la rappresentazione binaria e quella ottale è piuttosto semplice infatti se indichiamo con

$$o = 0, 1, 2, 3, 4, 5, 6, 7$$

i digits ottali e con

$$b = 0, 1$$

quelli binari si ha

$$\begin{aligned} & \ldots + b_8 2^8 + b_7 2^7 + b_6 2^6 + b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0 + \ldots \\ = \; & \ldots + \left(b_8 2^2 + b_7 2^1 + b_6 2^0\right) 2^6 + \left(b_5 2^2 + b_4 2^1 + b_3 2^0\right) 2^3 + \left(b_2 2^2 + b_1 2^1 + b_0 2^0\right) 2^0 + \ldots \\ = \; & \ldots + o_2 8^2 + o_1 8^1 + o_0 8^0 + \ldots \end{aligned}$$

poichè $2^6 = 8^2$, $2^3 = 8^1$ e così via e le espressioni tra parentesi sono la rappresentazione binaria di un digit ottale:

Ottale	0	1	2	3	4	5	6	7
Binario	000	001	010	011	100	101	110	111
Decimale	0	1	2	3	4	5	6	7

Di conseguenza:

Conversione Ottale → Binario: Si sostituisce ad ogni digit ottale la sua rappresentazione binaria. Ad esempio

$$\begin{array}{lcccccc} 472_8 & \Rightarrow & 4 & 7 & 2 & & \\ & & 100 & 111 & 010 & \Rightarrow & 100111010_2 \end{array}$$

Conversione Binario → Ottale: si raggruppano i digits binari in gruppi di tre partendo dal punto decimale, aggiungendo eventualmente digits 0 all'inizio e alla fine per formare i gruppi, e si sotituisce ad ogni gruppo il corrispondente digit ottale. Ad esempio

$$\begin{array}{lcccccccc} 11010.1011_2 & \Rightarrow & 011 & 010 & . & 101 & 100 & & \\ & & 3 & 2 & . & 5 & 4 & \Rightarrow & 32.54_8 \end{array}$$

Il primo e gli ultimi due 0 sono stati aggiunti per completare i gruppi.

A.4.4. Sistema esadecimale

Il sistema esadecimale o *hex* usa la stessa filosofia di base del sistema ottale che invece di usare $2^3 = 8$ simboli ne usa $2^4 = 16$:

0 1 2 3 4 5 6 7 8 9 A B C D E F

ed è quindi un sistema in base 16 con valore posizionale 16^k. Ciascun simbolo corrisponde al valore di 4 digits binari secondo la tavola

hex	**Binario**	**Decimale**		**hex**	**Binario**	**Decimale**
0	0000	0		8	1000	8

1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

per cui ad esempio

$$37A.15_8 = 3 \times 16^2 + 7 \times 16^1 + 10 \times 16^0 + 1 \times 16^{-1} + 5 \times 16^{-2} = 890.08203125_{10}$$

L'utilità del sistema esadecimale, come il sistema ottale, risiede nella semplicità con cui è possibile passare dalla rappresentazione binaria a quella ottale e viceversa. Il vantaggio del sistema esadecimale rispetto al sistema ottale è che il numero di simboli necessari per rappresentare un dato in rappresentazione binaria è diviso per 4 invece che 3, come nel sistema ottale.

Conversione Decimale → Esadecimale

Per trasformare un numero dall'usuale rappresentazione decimale a quella esadecimale si procede come per la trasformazione in rappresentazione binaria dividendo e moltiplicando per 16.

Conversione Esadecimale ↔ Binario

Per la trasformazione dalla rappresentazione esadecimale e quella binaria e viceversa si utilizzano regole simili a quella della trasformazione tra rappresentazione ottale e binaria binaria.

Conversione Esadecimale → Binario: Si sostituisce ad ogni digit esadecimale la sua rappresentazione binaria. Ad esempio

$$4C2_{16} \Rightarrow \begin{array}{ccc} 4 & C & 2 \\ 0100 & 1100 & 0010 \end{array} \Rightarrow 010011000010_2$$

Conversione Binario → Esadecimale: si raggruppano i digits binari in gruppi di quattro partendo dal punto decimale, aggiungendo eventualmente digits 0 all'inizio e alla fine per formare i gruppi, e si sotituisce ad ogni gruppo il corrispondente digit esadecimale. Ad esempio

$$11010.1011_2 \Rightarrow \begin{array}{cccc} 0001 & 1010 & . & 1011 \\ 1 & A & . & B \end{array} \Rightarrow 1A.B_{16}$$

I tre 0 sono stati aggiunti per completare il gruppo.

Conversione Esadecimale ↔ Ottale

La trasformazione tra rappresentazione esadecimale ed ottale si effettua facilmente passando per la rappresentazione binaria, ossia sostituendo a ciascun digit la sua rappresentazione binaria e raggruppandoli di nuovo in gruppi di tre o quattro digits a seconda della rappresentazione a cui si vuole passare.

A.5. Codici ASCII (Rev. 2.1)

- **Codice Decimale**

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

- **Codice Ottale**

000	NUL	001	SOH	002	STX	003	ETX	004	EOT	005	ENQ	006	ACK	007	BEL
010	BS	011	HT	012	NL	013	VT	014	NP	015	CR	016	SO	017	SI
020	DLE	021	DC1	022	DC2	023	DC3	024	DC4	025	NAK	026	SYN	027	ETB
030	CAN	031	EM	032	SUB	033	ESC	034	FS	035	GS	036	RS	037	US
040	SP	041	!	042	"	043	#	044	$	045	%	046	&	047	’
050	(	051	)	052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5	066	6	067	7
070	8	071	9	072	:	073	;	074	<	075	=	076	>	077	?
100	@	101	A	102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M	116	N	117	O
120	P	121	Q	122	R	123	S	124	T	125	U	126	V	127	W
130	X	131	Y	132	Z	133	[	134	\	135	]	136	^	137	_
140	‘	141	a	142	b	143	c	144	d	145	e	146	f	147	g
150	h	151	i	152	j	153	k	154	l	155	m	156	n	157	o
160	p	161	q	162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174	|	175	}	176	~	177	DEL

- **Codice Esadecimale**

00	NUL	01	SOH	02	STX	03	ETX	04	EOT	05	ENQ	06	ACK	07	BEL
08	BS	09	HT	0A	NL	0B	VT	0C	NP	0D	CR	0E	SO	0F	SI
10	DLE	11	DC1	12	DC2	13	DC3	14	DC4	15	NAK	16	SYN	17	ETB
18	CAN	19	EM	1A	SUB	1B	ESC	1C	FS	1D	GS	1E	RS	1F	US
20	SP	21	!	22	"	23	#	24	$	25	%	26	&	27	’
28	(	29	)	2a	*	2b	+	2c	,	2d	-	2e	.	2f	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3a	:	3b	;	3c	<	3d	=	3e	>	3f	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4a	J	4b	K	4c	L	4d	M	4e	N	4f	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5a	Z	5b	[	5c	\	5d	]	5e	^	5f	_
60	‘	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6a	j	6b	k	6c	l	6d	m	6e	n	6f	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7a	z	7b	{	7c	|	7d	}	7e	~	7f	DEL

A.6. Precedenza ed Associatività degli operatori (Rev. 2.1)

Ogni operatore in C ha un livello di precedenza e una regola di associatività. Le espressioni sono valutate raggruppando gli operandi a partire dagli operatori con livello di precedenza più alto. Se due operatori hanno la stessa precedenza, gli operandi sono raggruppati partendo da destra o da sinistra a seconda che l’associatività degli operatori sia da destra o da sinistra. Tutti gli operatori con lo stesso livello di precedenza hanno la stessa regola di associatività. L’ordine può essere modificato utilizzando le parentesi tonde “`()`”.

Simbolo	Operatore	Classe	Precedenza	Associatività
nome	variabili		16	n/a
`a[k]`	indice	postfisso	16	sinistra
`f(...)`	funzione	postfisso	16	sinistra
`.`	selezione diretta	postfisso	16	sinistra
`->`	selezione indiretta	postfisso	16	sinistra
`++ --`	incremento/decremento	postfisso	16	sinistra
`++ --`	incremento/decremento	prefisso	15	destra
`sizeof`	dimensione	unario	15	destra
`~`	not bit-a-bit	unario	15	destra
`!`	not logico	unario	15	destra
`- +`	segno negativo/positivo aritmetico	unario	15	destra
`&`	rerefenza	unario	15	destra
`*`	dereferenza	unario	15	destra
(*tipo*)	cast	unario	14	destra
`* / %`	moltiplicativi	binario	13	sinistra
`+ -`	additivi	binario	12	sinistra

Simbolo	Operatore	Classe	Precedenza	Associatività
`<< >>`	shift a sinistra/destra	binario	11	sinistra
`< > <= >=`	relazionali	binario	10	sinistra
`== !=`	uguaglianza	binario	9	sinistra
`&`	and bit-a-bit	binario	8	sinistra
`^`	xor bit-a-bit	binario	7	sinistra
`|`	or bit-a-bit	binario	6	sinistra
`&&`	and logico	binario	5	sinistra
`||`	or logico	binario	4	sinistra
`? :`	condizionale	ternario	3	destra
`= += -= *=`	assegnamento	binario	2	destra
`/= %= <<= >>=`	assegnamento	binario	2	destra
`&= ^= |=`	assegnamento	binario	2	destra
`,`	separazione	binario	1	sinistra

A.7. Modulo `yac_utils.c` (Rev. 2.0.1)

Questo modulo contiene le funzioni utilizzate spesso

```
#include "yac_utils.h"

int    *ivect(int n)   /* array di n int    */
float  *fvect(int n)   /* array di n float  */
double *dvect(int n)   /* array di n double */

int    **imatr(int n, int m)   -- array di n x m int
float  **fmatr(int n, int m)   -- array di n x m float
double **dmatr(int n, int m)   -- array di n x m double

FILE   *open_file(const char *path, const char *mode);
```

La funzione `open_file()` sostituisce la funzione `fopen()`. A differenza di quest'ultima interrompe l'esecuzione se l'apertura del file ha problemi.

Modulo: `yac_utils.c`

```
/*****************************************************************

- Descrizione : funzioni di utilizzo frequente

   #include "yac_utils.h"

   int    *ivect(int n)   -- array di n int
   float  *fvect(int n)   -- array di n float
   double *dvect(int n)   -- array di n double
```

```
   int    **imatr(int n, int m)   -- array di n x m int
   float  **fmatr(int n, int m)   -- array di n x m float
   double **dmatr(int n, int m)   -- array di n x m double

   FILE   *open_file(const char *path, const char *mode);

   sostituisce fopen()

- $Id: yac_utils.c v 1.3 08.11.04 AC
*************************************************************/
#include "yac_utils.h"

/* ============================================================ */
/*                  Routines di allocazione                     */
/* ============================================================ */

/* ----
 * ivect()
 */
extern int  *ivect(int n)
{
  int *v;

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

  if (v == NULL) {
    fprintf(stderr,"\n Cannot allocate memory \n");
    exit(EXIT_FAILURE);
  }
  return v;
}

/* ----
 * fvect()
 */
extern float *fvect(int n)
{
  float *v;

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

  if (v == NULL) {
    fprintf(stderr,"\n Cannot allocate memory \n");
    exit(EXIT_FAILURE);
  }

  return v;
}

/* ----
 * dvect()
 */
```

```
extern double *dvect(int n)
{
  double *v;

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

  if (v == NULL) {
    fprintf(stderr,"\n␣Cannot␣allocate␣memory␣\n");
    exit(EXIT_FAILURE);
  }

  return v;
}

/* ----
 * imatr()
 */
extern int **imatr(int n, int m)
{
  int **c;
  int   i;

  c = (int **) malloc(n * sizeof(int *));
  if ( c == NULL) {
    fprintf(stderr,"Memory␣allocation␣failure\n");
    exit(EXIT_FAILURE);
  }
  for (i =0; i < m; ++i) {
    c[i] = (int *) malloc(m * sizeof(int));
    if ( c[i] == NULL) {
      fprintf(stderr,"Memory␣allocation␣failure\n");
      exit(EXIT_FAILURE);
    }
  }
  return c;
}

/* ----
 * fmatr()
 */
extern float **fmatr(int n, int m)
{
  float **c;
  int     i;

  c = (float **) malloc(n * sizeof(float *));
  if ( c == NULL) {
    fprintf(stderr,"Memory␣allocation␣failure\n");
    exit(EXIT_FAILURE);
  }
  for (i =0; i < m; ++i) {
    c[i] = (float *) malloc(m * sizeof(float));
```

```
    if ( c[i] == NULL) {
      fprintf(stderr,"Memory allocation failure\n");
      exit(EXIT_FAILURE);
    }
  }
  return c;
}

/* ----
 * dmatr()
 */
extern double **dmatr(int n, int m)
{
  double **c;
  int      i;

  c = (double **) malloc(n * sizeof(double *));
  if ( c == NULL) {
    fprintf(stderr,"Memory allocation failure\n");
    exit(EXIT_FAILURE);
  }
  for (i =0; i < m; ++i) {
    c[i] = (double *) malloc(m * sizeof(double));
    if ( c[i] == NULL) {
      fprintf(stderr,"Memory allocation failure\n");
      exit(EXIT_FAILURE);
    }
  }
  return c;
}

/* ============================================================ */
/*                  Routines di I/O                             */
/* ============================================================ */
/* ----
 * open_file()
 */
extern FILE *open_file(const char *path, const char *mode)
{
  FILE *fa;

  if ( (fa = fopen(path, mode)) == NULL)
  {
    fprintf(stderr,"\n\nErrore in apertura di %s\n", path);
    exit(EXIT_FAILURE);
  }

  return fa;
}
```

Header: `yac_utils.h`

```
/*****************************************************************

- Descrizione : Header file per yac_utils.c

- $Id: yac_utils.h v 1.2 23.10.03 AC
*****************************************************************/
#include <stdlib.h>
#include <stdio.h>

/* ============================================================ */
/*                  Routines di allocazione                     */
/* ============================================================ */
extern int     *ivect(int n);
extern float   *fvect(int n);
extern double  *dvect(int n);
extern int    **imatr(int n, int m);
extern float  **fmatr(int n, int m);
extern double **dmatr(int n, int m);

/* ============================================================ */
/*                  Routines di I/O                             */
/* ============================================================ */
extern FILE *open_file(const char *path, const char *mode);
```

Part II.

Applicazioni