NoiStudiamo

DISPENSA DI TEORIA

Programmazione I in linguaggio C

Algoritmi, linguaggio C, puntatori, memoria, file, strutture dati, ordinamento e ricerca.

A CURA DI

Damiano La Pusata

Ottobre 2026

INDICE

Indice dei capitoli

I ventiquattro capitoli della dispensa, dagli algoritmi alla ricerca dicotomica. Nei listati il segno meno è stampato in forma tipografica: copiando il codice si digita il normale meno della tastiera.

1. Problemi e algoritmi 4

2. Diagrammi di flusso e programmazione strutturata 6

3. Linguaggi di programmazione e traduzione 10

4. Architettura e rappresentazione dell'informazione 14

5. Variabili, espressioni e assegnazioni 19

6. I tipi di dato del C 21

7. Operatori, espressioni e conversioni 23

8. Input e output di base 27

9. Il controllo del flusso 29

10. Le funzioni 33

11. Visibilità, durata e stack delle attivazioni 37

12. Numeri pseudocasuali e ricorsione 40

13. I puntatori 44

14. Array e aritmetica dei puntatori 46

15. Array multidimensionali 50

16. Le stringhe 51

17. Il qualificatore const e la memoria dinamica 54

18. Strutture, unioni ed enumerazioni 57

19. Argomenti del main e file 60

20. Tipi di dato astratti e insiemi 65

21. Liste concatenate 68

22. Pile e code 74

23. Algoritmi di ordinamento 78

24. Algoritmi di ricerca 83

CAPITOLO 1

Problemi e algoritmi

Che cosa significa risolvere un problema con un calcolatore: la definizione di algoritmo, le sue proprietà e il metodo con cui si progetta, dall'analisi del problema fino alla verifica con l'esecuzione a mano.

Problemi e algoritmi

CHE COS'È UN ALGORITMO

Dato un problema, un **algoritmo** è una sequenza di passi concepita per essere eseguita automaticamente da una macchina in modo da risolvere quel problema. In forma un po' più rigorosa: un algoritmo è una **sequenza ordinata e finita di passi** (azioni o istruzioni) che producono un **risultato ben determinato** in un **tempo finito**. Un problema che si può risolvere con un algoritmo si dice **computabile**.

L'esempio classico è la ricetta di cucina. Per preparare un risotto ai funghi si parte da dati iniziali (gli ingredienti), si segue una sequenza di passi (preparare i funghi, tostare il riso, cuocere aggiungendo brodo) e si ottiene un risultato (il piatto servito). Gli ingredienti sono l'**input**, la ricetta è l'**algoritmo**, il piatto è l'**output**.

Nella risoluzione di un problema con il calcolatore i ruoli sono divisi in modo netto. L'**uomo** è il vero **risolutore**: analizza il problema e concepisce l'algoritmo. La **macchina** è soltanto l'**esecutore**: riceve l'algoritmo codificato in un linguaggio che è in grado di interpretare, genera un **processo di esecuzione** e al termine produce l'output. Il procedimento complessivo è quindi: prendere i dati iniziali, concepire l'algoritmo e codificarlo, avviare l'esecutore, attendere la fine del lavoro e prelevare l'output.

LE PROPRIETÀ DI UN ALGORITMO

Perché una sequenza di passi sia davvero un algoritmo deve rispettare alcune proprietà. Le **azioni** devono essere **eseguibili e non ambigue**: espressioni come «abbastanza» o «a volontà», normali in una ricetta, non sono ammissibili, perché la macchina non sa interpretarle. L'algoritmo deve essere **deterministico**: fatto un passo, il successivo è uno e uno solo; le alternative sono ammesse, ma la scelta tra esse deve essere univoca, cioè decisa da una condizione verificabile. Deve avere un **numero finito di passi** e deve **terminare**.

Ogni singolo passo, inoltre, deve terminare entro un intervallo finito di tempo, produrre un **effetto osservabile** e produrre **sempre lo stesso effetto** ogni volta che viene eseguito a partire dalle stesse condizioni iniziali (stesso input, stessi valori iniziali delle variabili). È questa ripetibilità che permette di verificare un algoritmo eseguendolo a mano.

ATTENZIONE

Numero finito di passi non implica terminazione

Si consideri il procedimento seguente: «1. Si consideri un numero N; 2. scrivere N; 3. scrivere il numero successivo; 4. ripetere il passo precedente». Il **testo** dell'algoritmo è composto da quattro passi, quindi è finito, ma la sua **esecuzione** non termina mai, perché il passo 4 rimanda indefinitamente al passo 3. Finitezza della descrizione e terminazione dell'esecuzione sono due proprietà distinte, ed entrambe sono richieste.

Lo stesso problema può avere algoritmi diversi, e la scelta dipende dalla forma dell'input. Per cercare un nome in un elenco **non ordinato**, come un foglio di firme, non c'è alternativa a scorrerlo dall'inizio: è la **ricerca sequenziale**. Se l'elenco è **ordinato**, come un elenco telefonico, si può aprirlo a metà, capire se il nome sta prima o dopo e scartare ogni volta metà dei nomi rimasti: è la **ricerca dicotomica**, molto più veloce. Entrambe sono trattate nel capitolo 24, implementate in C.

DA RICORDARE

Che cos'è un algoritmo e quali proprietà deve avere?

Un algoritmo è una sequenza ordinata e finita di passi che, a partire da dati di input, produce un risultato ben determinato in un tempo finito; un problema risolvibile con un algoritmo si dice computabile. Le azioni devono essere eseguibili e non ambigue; l'algoritmo deve essere deterministico (dopo ogni passo il successivo è univocamente determinato), avere un numero finito di passi e terminare. Ogni passo deve concludersi in tempo finito, avere un effetto osservabile e dare sempre lo stesso effetto a parità di condizioni iniziali.

DALL'ALGORITMO AL PROGRAMMA

La macchina deve essere in grado di **comprendere** l'algoritmo, cioè di attribuire la giusta semantica a ogni passo ed eseguirlo correttamente. Per questo l'algoritmo va **codificato** in un linguaggio opportuno, in genere un linguaggio di **alto livello**: il risultato della codifica si chiama **programma**. Il programma in linguaggio di alto livello viene poi tradotto in **linguaggio macchina**, l'unico che il processore esegue davvero, e solo a quel punto la macchina può ricevere l'input e produrre l'output. La catena completa è: uomo → algoritmo → programma in linguaggio di alto livello → programma in linguaggio macchina → esecuzione.

Progettare un algoritmo

LE FASI DELLA PROGETTAZIONE

La progettazione segue uno schema fisso, utile per qualunque esercizio. Si parte dall'**analisi del problema**, che fornisce un nome, una breve descrizione di cosa si vuole fare e l'elenco dei **requisiti**, prevedendo tutti i casi possibili. Si passa poi alla **specifica funzionale**, che indica quali sono i **dati iniziali** (gli **ingressi** dell'algoritmo) e quale **risultato** si vuole in funzione degli ingressi (l'**uscita**).

Il terzo passo è individuare i **contenitori di dati**: l'algoritmo deve tenere traccia degli ingressi, del risultato finale e degli eventuali **risultati intermedi**. I contenitori degli ingressi e del risultato sono noti fin dall'inizio; quelli per i risultati intermedi dipendono dall'algoritmo, e capita spesso di aggiungerne «al volo» man mano che l'algoritmo prende forma. Ogni contenitore ha un **tipo** (per esempio intero), che stabilisce quali operazioni si possono applicare al suo contenuto. Questi contenitori diventeranno le **variabili** del programma.

Si descrive quindi l'**idea dell'algoritmo** a grandi linee, la si **raffina** aggiungendo dettagli, la si traduce in un **diagramma di flusso** e infine la si verifica con l'**esecuzione a mano** (tracing): si sceglie un input, si percorre il diagramma passo per passo e si annota il contenuto di ogni contenitore dopo ogni istruzione.

APPROCCIO TOP DOWN E BOTTOM UP

Nella ricetta del risotto alcuni passi, come «preparare il brodo vegetale» o «sfumare con il vino bianco», sono a loro volta dei **sottoproblemi**, che si possono scomporre in passi più elementari (lista degli ingredienti del brodo, tempo di cottura, filtraggio). Questa è l'idea della **scomposizione di un problema in sottoproblemi**.

Nell'**approccio top down** si costruisce prima una visione generale del problema senza scendere nei dettagli; poi ogni parte viene **raffinata per decomposizione**, aggiungendo dettagli, eventualmente con decomposizioni successive, finché la specifica non è abbastanza dettagliata da poter essere validata. Nell'**approccio bottom up** si parte invece dalle singole parti, specificate in dettaglio, e le si collega tra loro per formare componenti via via più grandi. Il bottom up puro è indicato quando si dispone già di molte componenti pronte, o di esperienza su problemi simili.

Nella pratica si adotta spesso un **approccio ibrido**: per stampare i nomi di persona presenti in un testo si procede top down (leggere il testo, memorizzare i nomi, stamparli), riusando bottom up funzioni già pronte, come una che legge una riga e una che la spezza in parole. In C ogni sottoproblema diventerà una **funzione**: un programma ben organizzato ha funzioni come `leggiParametri`, `leggiFile`, `scriviFile`, ciascuna responsabile di un sottoproblema.

CAPITOLO 2

Diagrammi di flusso e programmazione strutturata

Gli strumenti per descrivere un algoritmo in modo indipendente dal linguaggio: i diagrammi di flusso, la notazione lineare strutturata e il teorema che fonda la programmazione strutturata.

Diagrammi di flusso

La descrizione di un algoritmo precede la sua codifica, e va fatta in un linguaggio generale, indipendente dal linguaggio di programmazione: i due strumenti più usati sono i **diagrammi di flusso** e lo **pseudocodice** (nella forma della notazione lineare strutturata, capitolo 4).

BLOCCHI E CONNETTORI

Un diagramma di flusso descrive in modo grafico le azioni che compongono un algoritmo e il loro **flusso di esecuzione**. I **blocchi** rappresentano le azioni, i **connettori** (le frecce) stabiliscono in quale ordine vanno eseguite: è la posizione dei connettori, non quella dei blocchi sulla pagina, a determinare il flusso.

BLOCCO	FORMA	SIGNIFICATO
Inizio / Fine	ovale	punto di partenza e di arrivo dell'algoritmo; c'è un solo inizio
Input / Output	parallelogramma	«Leggi X», «Stampa Z»: acquisizione o comunicazione di dati
Istruzione	rettangolo	azione imperativa, tipicamente un assegnamento X ← espressione
Condizione	rombo	valuta un predicato; ha due uscite, VERO e FALSO
Connettore	freccia	indica il blocco da eseguire dopo

L'istruzione più importante è l'**assegnamento**, scritto `Variabile ← Espressione`. Una **variabile** è un'entità caratterizzata da un **nome** e da un **valore** che può cambiare nel tempo; un'**espressione** è una combinazione di operatori, costanti e variabili che dà luogo a un risultato. L'assegnamento valuta l'espressione a destra e copia il risultato nella variabile a sinistra: `Y ← Y/2 + Z` usa il valore attuale di Y per calcolarne uno nuovo. Il classico **scambio di due valori** richiede una variabile di appoggio: `Aux ← X`, `X ← Y`, `Y ← Aux`. Senza Aux, il primo assegnamento distruggerebbe il valore originale di X.

CONDIZIONI: PROPOSIZIONI E PREDICATI

Ci sono momenti in cui il flusso di esecuzione deve scegliere tra direzioni diverse, in base al verificarsi di una condizione: è l'**istruzione condizionale**. Una **proposizione** è un costrutto linguistico di cui si può affermare la verità o la falsità («il numero 2 è pari» è vera). Un **predicato** è una proposizione il cui valore di verità dipende dal valore di alcune variabili («X è minore di 30»). **Valutare** un predicato significa sostituire alle variabili i loro valori attuali e stabilire se è vero o falso; VERO e FALSO sono i valori **logici** o **booleani**. I predicati si scrivono in modo conciso con gli **operatori relazionali**: uguale (=), diverso (≠), minore (<), maggiore (>), minore o uguale (≤), maggiore o uguale (≥).

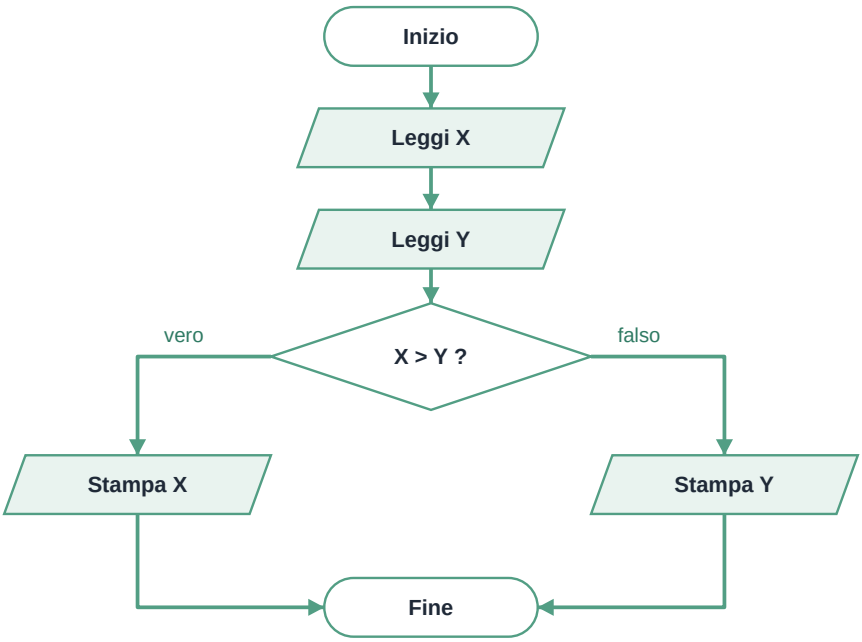


Diagramma di flusso: stampa del massimo tra due numeri

Il rombo del diagramma precedente valuta il predicato X > Y: se è vero si stampa X, altrimenti Y. Per stabilire se un numero è **pari o dispari** il rombo valuterebbe «il resto della divisione di X per 2 è 0?». Quando un connettore riporta il flusso a un blocco già eseguito si ottiene un **ciclo** (loop): nella ricetta, «far sobbollire a fuoco lento finché tutto l'alcol non sarà evaporato» è un ciclo che ripete la cottura finché la condizione «l'alcol è evaporato?» non diventa vera.

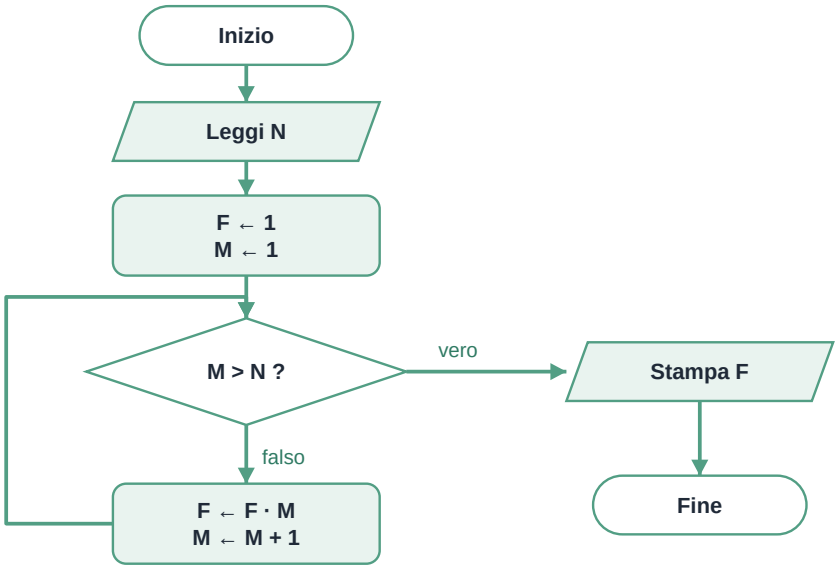


Diagramma di flusso del fattoriale N! con un ciclo

ESEMPIO SVOLTO

Esecuzione a mano del fattoriale con N = 3

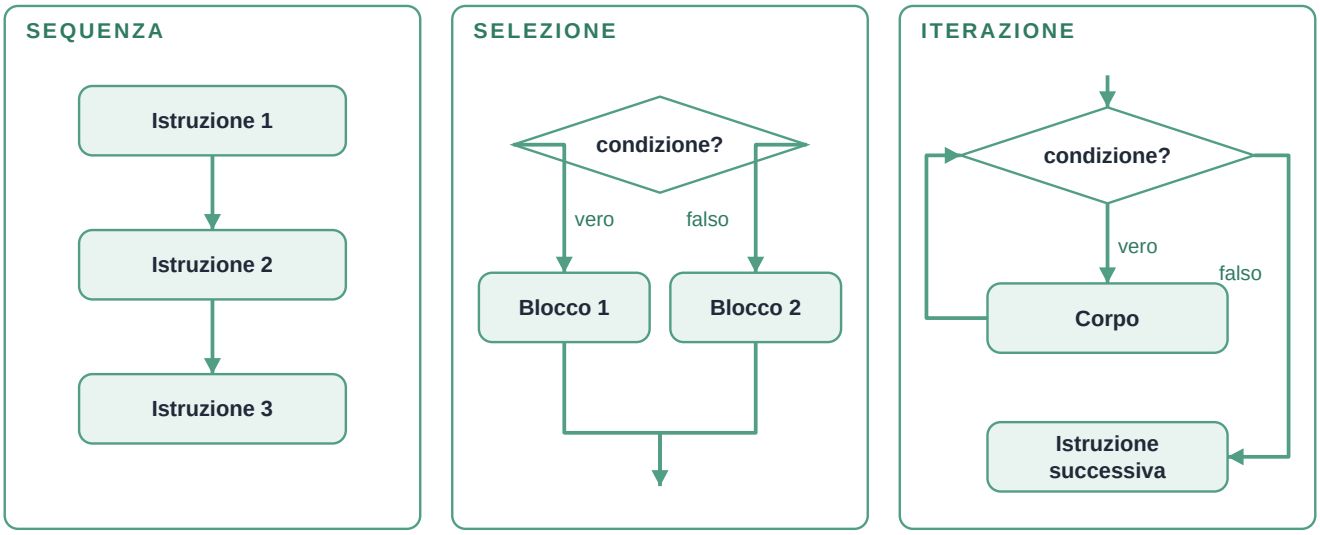
Dopo l'inizializzazione F = 1 e M = 1. Si valuta M > N, cioè 1 > 3: falso, quindi si esegue il corpo: F = 1·1 = 1, M = 2. Di nuovo 2 > 3 è falso: F = 1·2 = 2, M = 3. Ancora 3 > 3 è falso: F = 2·3 = 6, M = 4. Ora 4 > 3 è vero: si esce dal ciclo e si stampa F = 6, che è proprio 3!. Si noti che con N = 0 la condizione 1 > 0 è subito vera e si stampa F = 1, che è corretto perché 0! = 1.

Quando gli algoritmi diventano articolati, però, i diagrammi di flusso si rivelano **poco pratici**, soggetti a errori e **poco leggibili**: le frecce si intrecciano e diventa difficile seguire il flusso. L'alternativa è la notazione lineare strutturata.

Notazione lineare strutturata e teorema di Böhm e Jacopini

I TRE COSTRUTTI

La **notazione lineare strutturata** (NLS) descrive un algoritmo come testo, usando soltanto tre costrutti, ciascuno equivalente a uno schema di diagramma di flusso con **un solo ingresso e una sola uscita**. La **sequenza** è uno o più blocchi di operazioni eseguiti uno dopo l'altro. La **selezione** corrisponde al blocco condizionale: `If (condizione) Then blocco 1 Else blocco 2 End If`. L'**iterazione** corrisponde a un blocco condizionale più uno o più blocchi disposti in modo da formare un ciclo: `While (condizione) Do blocco End While`, che ripete il blocco finché la condizione è vera e, quando diventa falsa, prosegue con l'istruzione successiva.



I tre costrutti della programmazione strutturata

IL TEOREMA DI BÖHM E JACOPINI (1966)

Il **teorema di Böhm e Jacopini** afferma che **ogni algoritmo può essere costruito usando unicamente tre strutture di controllo: la sequenza, la selezione e l'iterazione**. Ogni altro tipo di istruzione di controllo, compresi i salti incondizionati (`goto`), può essere sostituito da una combinazione di questi tre schemi.

La conseguenza pratica è che in qualunque linguaggio di programmazione bastano costrutti che rappresentino le tre strutture per scrivere qualsiasi programma. I linguaggi ne offrono comunque altri per comodità: il C ha il ciclo `for`, il `do while`, lo `switch`, ma ognuno di essi si può riscrivere con sequenza, `if` e `while`. Il teorema è la base della **programmazione strutturata**, che rinuncia ai salti arbitrari in favore di blocchi con un solo ingresso e una sola uscita: il codice diventa leggibile dall'alto in basso, più facile da verificare e da modificare.

DA RICORDARE

Cosa afferma il teorema di Böhm e Jacopini e perché è importante?

Afferma che qualsiasi algoritmo si può esprimere usando soltanto tre strutture di controllo: sequenza, selezione (if then else) e iterazione (while). Ne segue che il salto incondizionato goto non è necessario e che ogni altro costrutto (for, do while, switch) è una comodità riconducibile ai tre. È il fondamento della programmazione strutturata, in cui ogni blocco ha un solo ingresso e una sola uscita, con vantaggi di leggibilità, verificabilità e manutenibilità del codice.

ESEMPI IN NLS

La **somma dei primi N numeri** usa un contatore i e un **accumulatore** S, entrambi inizializzati a 0 prima del ciclo: a ogni iterazione si incrementa i e lo si somma a S, finché i < N. Le **prime N+1 potenze di 2** usano la stessa struttura con P inizializzato a 1 e raddoppiato a ogni giro. Gli schemi «contatore» e «accumulatore» sono ricorrenti e conviene riconoscerli subito.

```
Inizio
  Leggi N
  i ← 0
  S ← 0
  While (i < N) Do
    i ← i + 1
    S ← S + i
  End While
  Stampa S
Fine
```

```
Inizio
  Leggi N
  M ← 0
  P ← 1
  While (M <= N) Do
    Stampa P
    P ← P · 2
    M ← M + 1
  End While
Fine
```

L'**algoritmo di Euclide per il M.C.D.** sfrutta la proprietà MCD(a, b) = MCD(b, a mod b): si sistema in MA il maggiore e in MB il minore, poi finché MB è diverso da 0 si calcola il resto r = MA % MB, si sposta MB in MA e r in MB. Quando MB vale 0, MA contiene il massimo comune divisore. Il **m.c.m.** si ottiene invece facendo crescere due multipli: MA parte da A e MB da B; finché sono diversi si aumenta di un passo il più piccolo (MA ← MA + A oppure MB ← MB + B). Il primo valore comune è il minimo comune multiplo.

```
Inizio
  Leggi A, B
  If (A < B) Then
    MA ← B;  MB ← A
  Else
    MA ← A;  MB ← B
  End If
  While (MB <> 0) Do
    r ← MA % MB
    MA ← MB
    MB ← r
  End While
  Stampa "MCD=", MA
Fine
```

```
Inizio
  Leggi A, B
  MA ← A
  MB ← B
  While (MA <> MB) Do
    If (MA > MB) Then
      MB ← MB + B
    Else
      MA ← MA + A
    End If
  End While
  Stampa "mcm=", MA
Fine
```

ESEMPIO SVOLTO

Tracing di MCD(21, 14) e mcm(3, 7)

MCD(21, 14). Dopo l'If: MA = 21, MB = 14. Primo giro: r = 21 % 14 = 7, MA = 14, MB = 7. Secondo giro: r = 14 % 7 = 0, MA = 7, MB = 0. La condizione MB <> 0 è falsa, si esce e si stampa MCD = 7.

mcm(3, 7). MA = 3, MB = 7. Si aumenta sempre il più piccolo: MA = 6, poi MA = 9; ora 9 > 7 e quindi MB = 14; poi MA = 12, MA = 15; 15 > 14 e quindi MB = 21; poi MA = 18, MA = 21. MA e MB sono uguali: si stampa mcm = 21.

CAPITOLO 3

Linguaggi di programmazione e traduzione

I livelli dei linguaggi di programmazione, la differenza tra sintassi e semantica, i principali paradigmi e il modo in cui un programma scritto in un linguaggio di alto livello diventa eseguibile.

Linguaggi di programmazione e paradigmi

TRE LIVELLI DI LINGUAGGIO

I linguaggi di programmazione si classificano in tre livelli. I **linguaggi macchina** (dal 1945) sono sequenze di bit direttamente eseguibili dal processore, specifiche di ogni architettura. I **linguaggi assembly** (dal 1950) sostituiscono i codici binari con abbreviazioni mnemoniche, ma restano legati all'architettura: un'istruzione assembly corrisponde a una istruzione macchina. I **linguaggi di alto livello** (dal 1950 in poi) evolvono verso **astrazione**, **semplificazione** e **somiglianza con il ragionamento umano**: una sola istruzione corrisponde a molte istruzioni macchina.

```
tot = var1 + var2;      // alto livello: un'unica istruzione

load  ACC, var1         // basso livello: carica var1 nell'accumulatore
add   ACC, var2         // somma var2 al contenuto dell'accumulatore
store tot, ACC          // copia il risultato in tot
```

SINTASSI E SEMANTICA

Un linguaggio di programmazione è definito da un insieme di regole formali. La **sintassi** (o grammatica) specifica tutti i simboli utilizzabili e come si devono comporre simboli e parole per formare istruzioni corrette. La **semantica** di un'istruzione ne definisce il **significato**, cioè l'effetto che produce. La **correttezza sintattica non è condizione sufficiente per la correttezza semantica**: un programma che deve sommare gli elementi di un array può compilare senza errori né warning e produrre un eseguibile funzionante, ma se il ciclo si ferma un elemento troppo presto il risultato è sbagliato. Il compilatore controlla la sintassi; la correttezza semantica è responsabilità del programmatore, e si verifica con il test e il debug.

DA RICORDARE

Qual è la differenza tra sintassi e semantica? Un programma che compila è corretto?

La sintassi stabilisce quali sequenze di simboli formano istruzioni valide; la semantica stabilisce cosa fanno. Un errore di sintassi (un punto e virgola mancante, una parentesi non chiusa) viene rilevato dal compilatore e impedisce di ottenere l'eseguibile. Un errore semantico, o logico, non viene rilevato: il programma compila e gira, ma produce un risultato diverso da quello voluto. Quindi un programma che compila è sintatticamente corretto, ma non necessariamente corretto.

PARADIGMI DI PROGRAMMAZIONE

Un **paradigma di programmazione** è la filosofia con cui si scrivono i programmi: caratterizza la metodologia (per esempio strutture di controllo oppure procedure) e il concetto stesso di computazione (istruzioni oppure funzioni matematiche). Ogni linguaggio si basa in genere su un paradigma.

La **programmazione imperativa** descrive il programma come una sequenza di istruzioni da «impartire» al calcolatore («leggi A», «assegna 5 a X»). La **programmazione strutturata** si basa sul teorema di Böhm e Jacopini e quindi sui tre costrutti sequenza, selezione e iterazione, eliminando il `goto`. La **programmazione procedurale** organizza il codice in blocchi ben delimitati, identificati da un nome ed eventuali argomenti (**funzioni** o procedure), ciascuno contenente istruzioni imperative e costrutti strutturati; qui compaiono i **side effect**: se una funzione riceve un parametro per indirizzo può modificare aree di memoria esterne a sé. La **programmazione modulare** suddivide il programma in **moduli** ben separati e indipendenti, ciascuno con un'**interfaccia** ben definita. La **programmazione orientata agli oggetti** (OOP) discende dalla procedurale, favorisce la modularità e include tutti gli aspetti precedenti.

Il **C** è un linguaggio **imperativo, strutturato e procedurale**, e con la compilazione separata (file .h e .c) supporta anche la **modularità**. Non è orientato agli oggetti: l'OOP sarà oggetto di Programmazione II.

APPROFONDIMENTO

Paradigmi non imperativi

Nella **programmazione funzionale** il flusso di esecuzione è una serie di valutazioni di funzioni matematiche, senza side effect: una funzione non modifica lo stato del mondo esterno. Ne derivano una verifica della correttezza più semplice e meno bug, al prezzo di un modello meno vicino al nostro modo di concepire i sistemi. Nella **programmazione logica** si usa la logica del primo ordine (un sistema formale di proposizioni e deduzioni) per descrivere la struttura logica del problema anziché il modo per risolverlo. Sono paradigmi meno diffusi, che esulano da questa dispensa.

Il problema della traduzione

PERCHÉ TRADURRE

Il processore esegue solo il linguaggio macchina, mentre noi vogliamo scrivere programmi in un linguaggio il più possibile vicino a quello umano: serve quindi una **traduzione**. Grazie alla traduzione i linguaggi hanno potuto evolvere verso sistemi simbolici più espressivi, scrivere programmi è diventato molto più agevole e si è ottenuta l'**indipendenza dalla piattaforma**: il programmatore scrive senza preoccuparsi dell'hardware sottostante. Un **traduttore** genera codice in linguaggio macchina a partire da codice scritto in un linguaggio di alto livello. Ne esistono due tipi: **compilatori** e **interpreti**.

COMPILATORI E INTERPRETI

Un **compilatore** prende in input l'intero **codice sorgente** P e lo traduce, **prima dell'esecuzione**, in un programma Q equivalente in linguaggio macchina (codice oggetto, poi eseguibile). C, C++, Pascal, Cobol e Fortran sono linguaggi compilati. Il compilatore deve «supportare» l'architettura di destinazione: un compilatore eseguito sull'architettura X produce codice per X, e per produrre codice destinato a un'altra architettura serve la **compilazione incrociata** (cross compilazione), con un'apposita toolchain.

Un **interprete** prende in input lo stesso codice sorgente, ma lavora un'istruzione alla volta: **traduce la singola istruzione**, **la esegue** e passa alla successiva. Traduzione ed esecuzione sono quindi intrecciate. Prolog, Lisp e Visual Basic sono esempi di linguaggi interpretati.

	COMPILATORE	INTERPRETE
Quando traduce	tutto il programma, prima di eseguirlo	un'istruzione alla volta, durante l'esecuzione
Prestazioni	(+) esecuzione nativa, veloce	(−) esecuzione più lenta
Risorse durante l'esecuzione	(+) il compilatore non occupa memoria durante l'esecuzione	(−) l'interprete occupa risorse sulla macchina di produzione
Installazione	(+) non invasivo: serve solo sulla macchina di sviluppo	(−) invasivo: va installato su ogni macchina di produzione
Portabilità	(−) l'eseguibile vale per una sola architettura	(+) l'interprete nasconde l'architettura

DA RICORDARE

Che differenza c'è tra un compilatore e un interprete? Il C a quale categoria appartiene?

Entrambi traducono codice di alto livello in linguaggio macchina. Il compilatore traduce l'intero sorgente una volta sola, prima dell'esecuzione, producendo un eseguibile autonomo: l'esecuzione è veloce e non richiede il compilatore sulla macchina di produzione, ma l'eseguibile è legato a un'architettura. L'interprete traduce ed esegue un'istruzione alla volta a ogni esecuzione: è più lento e deve essere installato dove il programma gira, ma rende il programma portabile. Il C è un linguaggio compilato; il compilatore più diffuso su Linux è gcc.

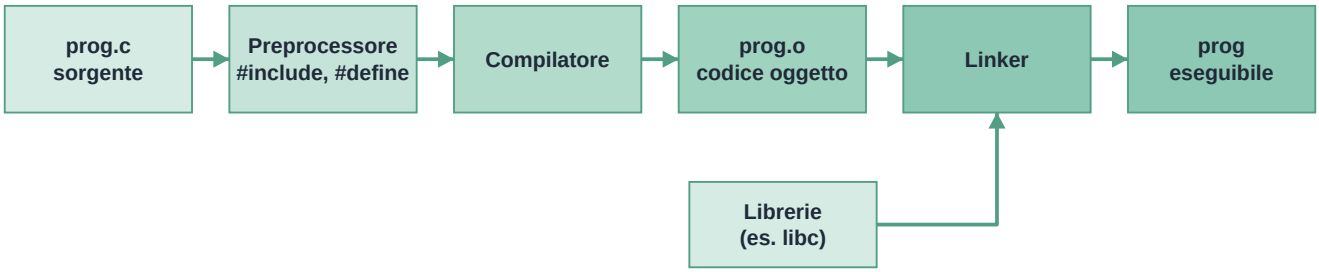
APPROFONDIMENTO

Il caso di Java: bytecode, JVM e JIT

Java (Sun Microsystems, 1995, oggi Oracle) è orientato agli oggetti ed è stato concepito per essere facilmente portabile. Adotta una soluzione mista: il sorgente viene **compilato** in **bytecode**, un linguaggio simile al linguaggio macchina ma indipendente dall'architettura; il bytecode viene poi **interpretato** dalla **Java Virtual Machine** (JVM), un programma in esecuzione sulla macchina di produzione. L'interpretazione comporta un overhead, che il compilatore **Just In Time** (JIT) riduce traducendo in codice macchina i metodi eseguiti più spesso: un metodo non viene compilato la prima volta che è caricato, ma dopo essere stato eseguito un certo numero di volte, perché anche il JIT consuma memoria e CPU.

DAL SORGENTE C ALL'ESEGUIBILE

Nel caso del C la «compilazione» comprende in realtà più fasi, che è utile distinguere fin da subito. Il **preprocessore** elabora le direttive che iniziano con `#`: `#include` inserisce il contenuto di un file header, `#define` sostituisce testualmente le costanti simboliche. Il **compilatore** vero e proprio traduce il risultato in **codice oggetto** (file `.o`): binario, ma non ancora eseguibile, perché le chiamate a funzioni di libreria come `printf` sono ancora riferimenti irrisolti. Il **linker** collega uno o più file oggetto con le **librerie** di sistema e produce l'**eseguibile**, che il sistema operativo carica in memoria ed esegue.



Le fasi della traduzione di un programma C

```
gcc –Wall prog.c –o prog     // preprocessore + compilazione + linking in un colpo solo
./prog                       // esecuzione

gcc –c prog.c                // solo compilazione: produce prog.o
gcc prog.o –o prog           // solo linking: da prog.o all'eseguibile
```

Lo sviluppo di un programma segue il ciclo **editing → compiling → running → debugging**: si scrive il sorgente con un editor, lo si compila correggendo gli errori segnalati, lo si esegue e, se il risultato non è quello atteso (errore semantico), si cerca l'errore con un **debugger** o con stampe di controllo, poi si ricomincia. Gli errori si distinguono quindi in **errori di compilazione** (sintattici, rilevati dal compilatore), **errori di linking** (per esempio una funzione dichiarata ma mai definita) ed **errori a tempo di esecuzione** o **logici**, che si manifestano solo eseguendo il programma.

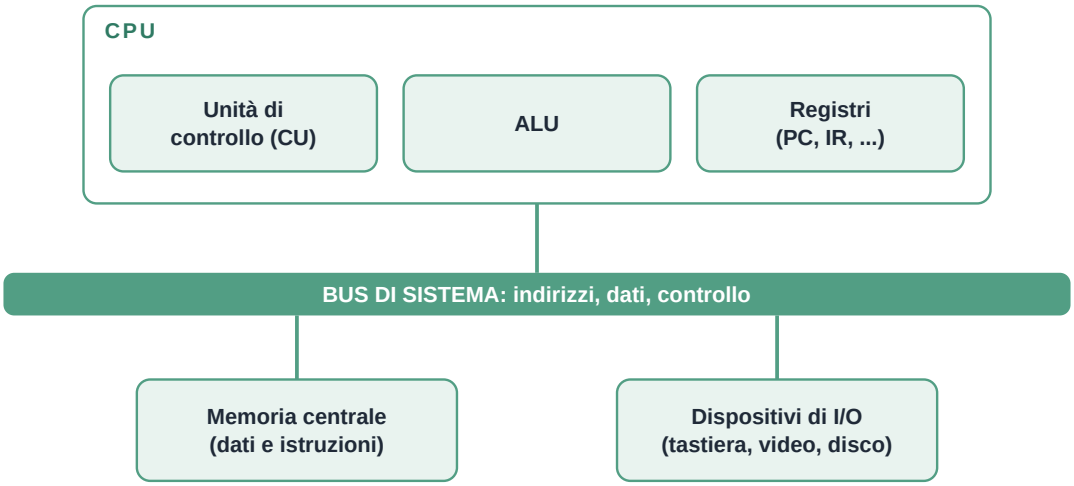
CAPITOLO 4

Architettura e rappresentazione dell'informazione

Il modello di calcolatore su cui girano i programmi e il modo in cui numeri, caratteri e dati multimediali vengono rappresentati con sequenze di bit, fino allo standard IEEE 754.

Cenni al modello di von Neumann

Quasi tutti i calcolatori moderni seguono l'architettura proposta da John von Neumann negli anni '40, la cui idea centrale è il **programma memorizzato**: **dati e istruzioni risiedono nella stessa memoria**, e il processore preleva da lì, una dopo l'altra, le istruzioni da eseguire. Cambiare programma significa quindi cambiare il contenuto della memoria, non ricablare la macchina.



Schema dell'architettura di von Neumann

I componenti sono quattro. L'**unità di elaborazione** (CPU) comprende l'**unità di controllo** (CU), che preleva e interpreta le istruzioni e coordina le altre componenti, l'**unità aritmetico logica** (ALU), che esegue operazioni aritmetiche e confronti, e i **registri**, piccole memorie velocissime interne alla CPU; tra questi il **Program Counter** (PC) contiene l'indirizzo della prossima istruzione e l'**Instruction Register** (IR) l'istruzione in corso. La **memoria centrale** (RAM) è una sequenza di **celle**, ciascuna identificata da un **indirizzo** numerico; è volatile, cioè perde il contenuto allo spegnimento. I **dispositivi di input/output** collegano la macchina al mondo esterno (tastiera, video, disco). Il **bus di sistema** collega tutti i componenti e trasporta indirizzi, dati e segnali di controllo.

La CPU lavora ripetendo all'infinito il **ciclo fetch, decode, execute**: nella fase di **fetch** preleva dalla memoria l'istruzione all'indirizzo contenuto nel PC e incrementa il PC; nella fase di **decode** interpreta l'istruzione; nella fase di **execute** la esegue, eventualmente leggendo o scrivendo dati in memoria. È esattamente il modello «memoria + unità di elaborazione» richiamato dal programma: un esecutore che legge istruzioni e dati da una memoria indirizzata.

Questo modello spiega già alcuni concetti che ritroveremo in C. Una **variabile** è il nome che il programma dà a una o più celle di memoria: ha un **indirizzo** (dove si trova) e un **contenuto** (il valore). L'operatore `&` del C restituisce proprio l'indirizzo di una variabile, ed è su questa distinzione tra indirizzo e contenuto che si basano i **puntatori**.

Rappresentazione digitale dell'informazione

All'interno del calcolatore ogni informazione (numeri, testo, immagini, suoni) è rappresentata come una sequenza di **bit**, cifre binarie che valgono 0 o 1. Otto bit formano un **byte**, l'unità minima indirizzabile della memoria. Con n bit si rappresentano 2^n configurazioni diverse: con 8 bit 256, con 16 bit 65 536, con 32 bit circa 4,3 miliardi. Il modo in cui una configurazione di bit viene interpretata dipende dal **tipo** del dato: la stessa sequenza può essere un intero, un carattere o parte di un numero reale.

BOOLEANI

Un valore **booleano** ha solo due valori possibili, vero e falso, e in linea di principio basta un bit per rappresentarlo. In C non esiste un tipo booleano primitivo nel senso classico: qualunque espressione che vale **0 è falsa** e qualunque valore **diverso da 0 è vero**; i confronti restituiscono 1 se veri e 0 se falsi. Includendo `<stdbool.h>` si dispone del tipo `bool` e delle costanti `true` e `false`, che occupano comunque almeno un byte, perché il byte è la più piccola unità indirizzabile.

SISTEMI DI NUMERAZIONE POSIZIONALI

Nei sistemi **posizionali** il valore di una cifra dipende dalla posizione che occupa: ogni posizione vale una potenza della **base**. Il sistema decimale ha base 10, il binario base 2: $102_{10} = 1\cdot10^2 + 0\cdot10^1 + 2\cdot10^0$, e analogamente $10101010_2 = 1\cdot2^7 + 0\cdot2^6 + 1\cdot2^5 + 0\cdot2^4 + 1\cdot2^3 + 0\cdot2^2 + 1\cdot2^1 + 0\cdot2^0 = 170_{10}$. Per la parte frazionaria le posizioni a destra della virgola valgono potenze negative: $101.0101_2 = 5 + (0\cdot2^{-1} + 1\cdot2^{-2} + 0\cdot2^{-3} + 1\cdot2^{-4}) = 5 + 0{,}3125 = 5{,}3125_{10}$.

In generale, una rappresentazione $c_m\ldots c_1c_0$ in base B vale $N = c_m\cdot B^m + \ldots + c_1\cdot B^1 + c_0\cdot B^0$: è l'algoritmo di **conversione da base B a base 10** che parte da risultato = 0 e, scorrendo le cifre dalla più significativa alla meno significativa, somma a ogni passo cifra × B^posizione. Applicato a 1101_2 dà 8 + 4 + 0 + 1 = 13_{10}.

Per la **conversione da base 10 a base 2** di un intero si divide ripetutamente per 2 annotando i resti; i **resti letti in ordine inverso** formano il numero binario: il primo resto è il bit meno significativo (più a destra), l'ultimo il più significativo. Per la **parte frazionaria** si moltiplica ripetutamente per 2: la parte intera di ogni prodotto (il «riporto», 0 o 1) è la cifra successiva, e i riporti si leggono **nell'ordine in cui sono ottenuti**; ci si ferma quando la parte frazionaria diventa 0.

ESEMPIO SVOLTO

Conversioni da base 10 a base 2

136_{10}. 136:2 = 68 r 0; 68:2 = 34 r 0; 34:2 = 17 r 0; 17:2 = 8 r 1; 8:2 = 4 r 0; 4:2 = 2 r 0; 2:2 = 1 r 0; 1:2 = 0 r 1. Resti dal basso verso l'alto: **$136_{10} = 10001000_2$**. Verifica: $2^7 + 2^3 = 128 + 8 = 136$.

$28{,}125_{10}$. Parte intera: 28 = 11100_2. Parte frazionaria: 0,125·2 = 0,25 → 0; 0,25·2 = 0,5 → 0; 0,5·2 = 1,0 → 1, e la parte frazionaria è 0: fine. Quindi 0,125 = $0{,}001_2$ e **$28{,}125_{10} = 11100{,}001_2$**.

$17{,}55_{10}$ con 8 cifre frazionarie. Parte intera 10001_2. Parte frazionaria: 0,55 → 1,10 (1); 0,10 → 0,20 (0); 0,20 → 0,40 (0); 0,40 → 0,80 (0); 0,80 → 1,60 (1); 0,60 → 1,20 (1); 0,20 → 0,40 (0); 0,40 → 0,80 (0)… e da qui la sequenza 0011 si ripete all'infinito. Troncando all'ottava cifra: **$17{,}55_{10} \approx 10001{,}10001100_2$**.

L'ultimo esempio mostra un fatto fondamentale: **alcuni numeri con parte frazionaria non si possono rappresentare con un numero finito di cifre**. Succede anche in base 10 (1/3 = 0,333…), ma in base 2 colpisce numeri che in decimale sembrano innocui: $0{,}1_{10} = 0{,}000110011\ldots_2$ periodico, così come $4{,}35_{10} = 100{,}01011001100\ldots_2$. Con un numero finito di bit questi valori possono essere solo **approssimati**.

Accanto a binario e decimale si usano spesso le basi **8 (ottale)** e **16 (esadecimale)**, perché un gruppo di 3 bit corrisponde a una cifra ottale e un gruppo di 4 bit a una cifra esadecimale (cifre da 0 a 9 e da A a F): 10001000_2 = 1000 1000 = 0x88. In C le costanti esadecimali si scrivono con il prefisso `0x` e quelle ottali con lo `0` iniziale; `printf` le stampa con `%x` e `%o`.

NUMERI INTERI

Nel calcolatore i numeri si rappresentano con sequenze di bit di lunghezza fissata (8, 16, 32, 64 bit). Per gli **interi senza segno** tutti gli n bit esprimono il valore assoluto, e l'intervallo rappresentabile è $[0, 2^n - 1]$: con 16 bit [0, 65 535], con 32 bit [0, 4 294 967 295].

Per gli **interi con segno** un bit è dedicato al segno, quindi il modulo massimo rappresentabile si dimezza. La rappresentazione usata in pratica dai processori, e quindi dagli interi del C, è il **complemento a due**: il bit più significativo ha peso negativo -2^{n-1} e gli altri il consueto peso positivo. L'intervallo è asimmetrico, **$[-2^{n-1}, 2^{n-1} - 1]$**: con 8 bit da −128 a +127, con 16 bit da −32 768 a +32 767, con 32 bit da −2 147 483 648 a +2 147 483 647. Lo zero ha una sola rappresentazione (tutti bit a 0) e −1 è rappresentato da tutti bit a 1.

ESEMPIO SVOLTO

Complemento a due su 8 bit

Per ottenere −5: si scrive +5 in binario, 00000101; si **invertono tutti i bit**, 11111010; si **aggiunge 1**, 11111011. Verifica con i pesi: −128 + 64 + 32 + 16 + 8 + 2 + 1 = −5. Il vantaggio del complemento a due è che somma e sottrazione usano lo stesso circuito: 5 + (−5) = 00000101 + 11111011 = 1 00000000, e il riporto oltre l'ottavo bit si scarta, dando 0.

All'interno dell'intervallo **tutti gli interi sono rappresentati esattamente**: niente approssimazioni e niente underflow. Il rischio è l'**overflow**: il risultato di un'espressione supera, in valore assoluto, il massimo rappresentabile. Negli interi senza segno il valore «ricomincia» da 0; negli interi con segno, sommando 1 a +2 147 483 647 si ottiene tipicamente −2 147 483 648, e per il linguaggio C l'overflow con segno è un comportamento non definito. Per non sprecare memoria e non incorrere in overflow bisogna scegliere la rappresentazione adatta: quanti bit, e se serve il segno. Per valori dell'ordine di 10^{10}, che superano i 32 bit, si usano i `long` a 64 bit.

NUMERI REALI: LA VIRGOLA MOBILE E LO STANDARD IEEE 754

I numeri reali si rappresentano nel formato in **virgola mobile** (floating point), una forma compatta derivata dalla notazione scientifica. In base 10: $96{,}103 = 0{,}96103 \times 10^2$; $2{,}96 = 0{,}296 \times 10^1 = 29{,}6 \times 10^{-1}$. Il numero 0,96103 si chiama **mantissa** (o significando), 10 è la **base**, 2 è l'**esponente**. Gli ultimi due esempi rappresentano lo stesso numero spostando la virgola e correggendo l'esponente: la virgola è «mobile», da cui il nome.

Lo standard **IEEE 754** fissa il formato binario: **1 bit di segno** s, un campo **esponente** E e un campo **mantissa** M, in modo che il numero valga $N = (-1)^s \times 1{,}M \times 2^e$. In **singola precisione** (32 bit, il `float` del C) l'esponente occupa 8 bit e la mantissa 23; in **doppia precisione** (64 bit, il `double`) l'esponente occupa 11 bit e la mantissa 52.

FORMATO	SEGNO	ESPONENTE	MANTISSA	BIAS K	CIFRE DECIMALI SIGNIFICATIVE
singola (float, 32 bit)	1 bit	8 bit	23 bit	127	circa 6 o 7 (FLT_DIG = 6)
doppia (double, 64 bit)	1 bit	11 bit	52 bit	1023	circa 15 o 16 (DBL_DIG = 15)

La **mantissa** è memorizzata in **forma normalizzata**: si moltiplica o divide il numero binario per una potenza di 2 finché resta una sola cifra prima della virgola, che è necessariamente 1. Poiché quell'1 c'è sempre, **non viene memorizzato** (bit implicito) e si guadagna un bit di precisione: nei 23 bit si scrive solo la parte dopo la virgola.

L'**esponente** non è memorizzato con segno, ma **con un bias** k: si memorizza E = e + k, dove e è l'esponente effettivo. In singola precisione k = 127; gli 8 bit darebbero 256 combinazioni, ma i valori E = 0 ed E = 255 sono riservati ai valori speciali, quindi E va da 1 a 254 ed e da −126 a +127. Usare il bias invece del segno permette di confrontare due numeri positivi confrontando i loro bit come se fossero interi senza segno.

ESEMPIO SVOLTO

Codifica di −113,25 in singola precisione

1. Binario del valore assoluto. 113 = 1110001_2 e 0,25 = $0{,}01_2$, quindi 113,25 = $1110001{,}01_2$.

2. Normalizzazione. Si sposta la virgola di 6 posizioni a sinistra: 1110001,01 = 1,11000101 × 2^6. L'esponente effettivo è e = 6.

3. Mantissa. Si scartano l'1 implicito e la virgola e si completa a 23 bit con zeri: M = 11000101000000000000000.

4. Esponente. E = e + 127 = 133 = 10000101_2.

5. Segno. Il numero è negativo: s = 1.

Risultato: 1 | 10000101 | 11000101000000000000000, cioè 0xC2E28000 in esadecimale.

Il percorso inverso. Data la sequenza 0 | 10000001 | 0111000…0: s = 0 (positivo); E = 129, quindi e = 129 − 127 = 2; mantissa $1{,}0111_2$. Il valore è $1{,}0111_2 \times 2^2 = 101{,}11_2 = 5{,}75_{10}$.

Il formato in virgola mobile introduce tre problemi che con gli interi non esistono. Il primo è l'**approssimazione**, che nasce in due casi: quando il numero non è rappresentabile con un numero finito di cifre binarie (4,35 o 0,1) e la sua rappresentazione viene troncata; e quando il numero ha **più cifre significative** di quante ne contenga la mantissa. 9 876 543,25 = 1,00101101011010000011111101 × 2^{23} richiede 25 bit di mantissa contro i 23 disponibili, quindi viene memorizzato in modo approssimato pur essendo ben dentro l'intervallo rappresentabile. La **precisione** p di un formato è il numero di cifre significative che può rappresentare: circa 6 cifre decimali per il `float` e 15 per il `double`.

Il secondo problema è l'**overflow**: il valore assoluto del risultato supera il più grande numero rappresentabile, circa 3,4 × 10^{38} per il `float` e 1,8 × 10^{308} per il `double`. Il terzo è l'**underflow**, tipico della virgola mobile: il risultato è **più piccolo in valore assoluto** del più piccolo numero rappresentabile diverso da zero (circa 1,2 × 10^{-38} per il `float` in forma normalizzata), e viene arrotondato a zero.

Le configurazioni riservate dell'esponente codificano i **valori speciali**. **±Inf** (infinito) è il risultato di un overflow o della divisione in virgola mobile di un numero non nullo per zero (x/0). **NaN** (*Not a Number*) è il risultato delle forme indeterminate, come 0/0, ∞/∞ o +Inf − Inf. **±0**: esistono due zeri, con segno, che sono il risultato degli underflow positivi o negativi.

ATTENZIONE

Mai confrontare due float con ==

Poiché 0,1 e 0,2 non sono rappresentabili esattamente, in un programma C l'espressione `0.1 + 0.2 == 0.3` vale **falso**: la somma dei due valori approssimati differisce di pochissimo dalla rappresentazione approssimata di 0,3. Per confrontare numeri in virgola mobile si controlla che la differenza sia minore di una tolleranza, per esempio `fabs(a − b) < 1e−9`. Per lo stesso motivo un contatore di ciclo non dovrebbe mai essere un `float`.

DA RICORDARE

Come si rappresentano interi e reali nel calcolatore? Che differenza c'è tra overflow e underflow?

Gli interi si rappresentano in binario su un numero fisso di bit, senza segno nell'intervallo [0, 2^n − 1] o con segno in complemento a due nell'intervallo [-2^{n-1}, 2^{n-1} − 1]; dentro l'intervallo sono esatti e l'unico rischio è l'overflow. I reali si rappresentano in virgola mobile secondo IEEE 754, con segno, esponente con bias e mantissa normalizzata a bit implicito, su 32 bit (float, circa 6 cifre di precisione) o 64 bit (double, circa 15 cifre). L'overflow si ha quando il risultato è troppo grande in valore assoluto per essere rappresentato; l'underflow, proprio della virgola mobile, quando è troppo piccolo in valore assoluto e viene arrotondato a zero. In virgola mobile si aggiungono gli errori di approssimazione, dovuti a numeri non rappresentabili con un numero finito di bit o con più cifre significative della precisione disponibile.

CARATTERI E STRINGHE

I **caratteri** si rappresentano associando a ciascuno un numero tramite una tabella di codifica. La più semplice è il codice **ASCII** a 7 bit, che codifica 128 caratteri: i codici da 0 a 31 sono caratteri di controllo (come il ritorno a capo, 10), poi seguono spazio, punteggiatura, cifre, lettere. Valgono la pena di essere ricordati alcuni valori: **'0' = 48**, **'A' = 65**, **'a' = 97**. Cifre e lettere sono consecutive, quindi `'7' - '0'` vale 7 e una lettera minuscola si ottiene dalla maiuscola sommando 32. In C il tipo `char` occupa un byte e contiene proprio il codice numerico: `c = 'K'` e `c = 75` sono equivalenti.

Una **stringa** è una sequenza di caratteri. In C non è un tipo a sé, ma un **array di** `char` **terminato dal carattere nullo** `'\0'` (codice 0), che segna dove la stringa finisce: la stringa "ciao" occupa quindi **5 byte**, quattro per le lettere e uno per il terminatore. Per questo un array destinato a contenere stringhe di al più LUN caratteri si dichiara di dimensione LUN + 1. Le stringhe sono trattate in dettaglio nel capitolo 16.

DATI MULTIMEDIALI

APPROFONDIMENTO

Immagini, suoni e video in bit

Anche i dati multimediali sono sequenze di numeri. Un'**immagine** raster è una griglia di **pixel**; in un'immagine a colori ogni pixel è descritto da tre intensità (rosso, verde, blu), tipicamente un byte ciascuna, cioè 24 bit per pixel: un'immagine 1920 × 1080 occupa così circa 6,2 MB senza compressione. In un'immagine in scala di grigi basta un byte per pixel. Un **suono** si digitalizza **campionando** l'onda a intervalli regolari e memorizzando ogni campione come un intero: la qualità CD usa 44 100 campioni al secondo, 16 bit per campione e 2 canali. Un **video** è una sequenza di immagini (frame) accompagnata da un audio. Le dimensioni elevate spiegano perché questi dati vengono quasi sempre **compressi** (JPEG, MP3, MP4).

CAPITOLO 5

Variabili, espressioni e assegnazioni

I concetti di variabile, costante, espressione e assegnamento, che fanno da ponte tra la descrizione di un algoritmo e il linguaggio C.

Variabili

Una **variabile** è un **contenitore di dati identificato da un nome** all'interno del programma. Corrisponde a un certo **indirizzo** nella memoria del calcolatore e le è associato un **tipo**, che deve essere adeguato all'informazione da memorizzare e che stabilisce quanta memoria occupa e quali operazioni ammette. Una variabile ha quindi quattro attributi: **nome**, **tipo**, **indirizzo**, **valore**. È la traduzione in C dei «contenitori di dati» della progettazione e delle celle di memoria del modello di von Neumann.

```
int j;                              // definizione senza inizializzazione
int numero_ordini = 10;             // definizione con inizializzazione
int totale = prezzo * quantita;     // inizializzazione con altre variabili
int v1, v2 = 3, v3;                 // solo v2 e' inizializzata
char c = 'a';                       // variabile di tipo carattere
```

Il nome deve **descrivere il contenuto** della variabile. Le regole sono precise: il nome inizia con una **lettera o un underscore** (_), i caratteri successivi possono essere lettere, cifre o underscore, **non sono ammessi spazi** e non si possono usare le **parole riservate** del linguaggio (`int`, `double`, `while`…). Il C è **case sensitive**: `var` e `Var` sono due variabili diverse. Il punto e virgola termina l'istruzione di definizione.

Il C **non obbliga a inizializzare** le variabili, ma una variabile locale non inizializzata contiene un valore indeterminato («spazzatura»), cioè quello che si trovava già in quella zona di memoria. Usarla prima di averle assegnato un valore è un errore semantico che il compilatore non sempre segnala. Anche l'**ordine** conta: se si scrive `int b; int volume = a * b * h; b = 15;`, nel calcolo del volume b non vale ancora 15, perché l'assegnamento arriva dopo.

Letterali e costanti

Un **valore letterale** è un elemento del programma che rappresenta direttamente un valore. Il suo tipo è deciso dalla forma: `-6` è un `int`; `0.5` è un `double`; `0.5f` è un `float`; `1E6` è un `double` in notazione esponenziale (1×10^6); `'c'` è un `char`; `"acqua"` è una stringa. Il separatore decimale è il **punto**: `10,456` non è un numero e `3 1/2` non è un letterale valido (si scrive `3.5`). Scrivere `int codice = "AA10";` è un errore, perché si assegna una stringa a un intero.

Quando un valore non deve cambiare si usa una **costante**. Con `const int VALORE_BANCONOTA = 10;` la variabile va inizializzata nella definizione e il compilatore segnala come errore ogni tentativo di modificarla. In alternativa si usa una **costante simbolica** del preprocessore, `#define MAX 100`, senza tipo, senza `=` e senza punto e virgola, che il preprocessore sostituisce testualmente prima della compilazione. Le costanti migliorano la leggibilità e riducono gli errori: se il valore della banconota passa da 10 a 20 basta cambiare una riga, invece di cercare tutte le occorrenze del numero 10 nel codice, dove potrebbe avere anche significati diversi.

I **commenti** sono testo ignorato dal compilatore e servono a chi legge il codice: `//` commenta fino a fine riga, `/* … */` racchiude un commento su più righe.

Espressioni e assegnamento

Un'**espressione** combina operandi (variabili, letterali, chiamate di funzione) e operatori, e la sua valutazione produce un valore di un certo tipo. L'**assegnamento** `variabile = espressione;` valuta l'espressione a destra e ne copia il valore nella variabile a sinistra. Va letto come «diventa», non come un'uguaglianza matematica: `var = var + 10;` significa (1) leggi il contenuto di var, (2) somma 10 al valore letto, (3) copia il risultato in var. In matematica l'equazione var = var + 10 non avrebbe soluzione; in un programma è l'aggiornamento più comune che ci sia.

Ciò che può comparire a sinistra di un assegnamento si chiama **lvalue**: deve indicare una zona di memoria modificabile, cioè una variabile. Ciò che può comparire a destra è un **rvalue**, un qualunque valore. `x = 1;` è valido; `5 = x;` e `h + k = 2;` sono errori di compilazione, perché né il letterale 5 né il risultato di h + k sono locazioni di memoria in cui scrivere. Oltre all'assegnamento, il valore di una variabile può cambiare con gli operatori di **incremento e decremento**: `a++` e `++a` equivalgono a `a = a + 1`, `b--` e `--b` a `b = b - 1`. La differenza tra forma prefissa e postfissa sarà vista con gli operatori del C.

In C anche l'assegnamento è un'espressione, il cui valore è il valore assegnato: per questo si può scrivere `a = b = 0;`, che assegna 0 a b e poi il valore dell'espressione (0) ad a. Questa idea, «l'esecuzione dei comandi come valutazione di espressioni», è ripresa nel capitolo 7.

ATTENZIONE

Assegnamento (=) e confronto (==)

In C = è l'**assegnamento** e == il **confronto di uguaglianza**. Scrivere `if (x = 5)` non confronta x con 5: assegna 5 a x, e l'espressione vale 5, cioè «vero», quindi il ramo if viene eseguito sempre. Il compilatore lo accetta, perché sintatticamente è corretto, e al massimo emette un warning con `-Wall`. È un esempio concreto di programma sintatticamente corretto e semanticamente sbagliato.

DA RICORDARE

Che cos'è una variabile? Che differenza c'è tra una variabile e una costante?

Una variabile è un contenitore di dati identificato da un nome, a cui corrispondono un tipo, un indirizzo in memoria e un valore che può cambiare durante l'esecuzione tramite assegnamenti o incrementi. Il tipo stabilisce quanta memoria occupa e quali operazioni sono ammesse. Una costante è un valore che non può cambiare: in C la si ottiene con il qualificatore const, che richiede l'inizializzazione nella definizione e fa segnalare al compilatore ogni tentativo di modifica, oppure con la direttiva #define, che il preprocessore sostituisce testualmente prima della compilazione. Le costanti rendono il codice più leggibile e facile da modificare.

CAPITOLO 6

I tipi di dato del C

I tipi primitivi del C, la memoria che occupano, gli intervalli di valori che rappresentano e il modo di definire costanti con nome.

I tipi primitivi

Il **tipo** di una variabile stabilisce quanta memoria occupa, come vanno interpretati i suoi bit, quale intervallo di valori può contenere e quali operazioni ammette. Il C ha due famiglie di tipi primitivi: gli **interi** (`char`, `short`, `int`, `long`, `long long`, ciascuno anche nella versione `unsigned`, senza segno) e i **reali in virgola mobile** (`float`, `double`, `long double`). Lo standard fissa solo dimensioni minime e la relazione char ≤ short ≤ int ≤ long ≤ long long; le dimensioni reali dipendono dal compilatore e dall'architettura. La tabella riporta i valori di gcc su Linux a 64 bit, la piattaforma più comune.

TIPO	BYTE	INTERVALLO (GCC, LINUX 64 BIT)	PRINTF	SCANF
char	1	−128 … 127 (un carattere)	%c, %d	%c
unsigned char	1	0 … 255	%c, %u	%c
short	2	−32 768 … 32 767	%hd	%hd
unsigned short	2	0 … 65 535	%hu	%hu
int	4	−2 147 483 648 … 2 147 483 647	%d	%d
unsigned int	4	0 … 4 294 967 295	%u	%u
long	8	-2^{63} … $2^{63} - 1$ (circa $\pm 9{,}2 \cdot 10^{18}$)	%ld	%ld
unsigned long	8	0 … $2^{64} - 1$	%lu	%lu
long long	8	come long	%lld	%lld
float	4	$\pm 3{,}4 \cdot 10^{38}$, circa 6 cifre significative	%f, %e, %g	%f
double	8	$\pm 1{,}8 \cdot 10^{308}$, circa 15 cifre significative	%f, %e, %g	%lf
long double	16	precisione estesa	%Lf	%Lf

Il `char` è un intero a tutti gli effetti, grande un byte, che di solito contiene il codice ASCII di un carattere: `'A' + 2` vale 67, cioè `'C'`. Il tipo `void` indica «nessun valore»: si usa per le funzioni che non restituiscono nulla e, come `void *`, per i puntatori generici restituiti da funzioni come `malloc`. Il tipo `bool` con i valori `true` e `false` richiede `#include <stdbool.h>`. Su Windows a 64 bit `long` è grande 4 byte e non 8: è un esempio di come le dimensioni non siano garantite.

sizeof, limits.h e float.h

L'operatore `sizeof` restituisce il numero di byte occupati da un tipo o da una variabile: `sizeof(int)` vale 4, `sizeof(double)` vale 8. Si usa come una funzione, ma è un operatore valutato in fase di compilazione, e il suo risultato ha tipo `size_t` (un intero senza segno), che si stampa con `%zu`. Gli header `limits.h` e `float.h` definiscono i limiti dei tipi: `INT_MIN`, `INT_MAX`, `LONG_MAX`, `UINT_MAX`, `CHAR_MAX` per gli interi; `FLT_MAX`, `DBL_MAX`, `FLT_MIN` (il più piccolo float positivo normalizzato) e `FLT_DIG`, `DBL_DIG` (le cifre significative) per i reali.

Conoscere dimensioni e intervalli serve per tre motivi. Quando si alloca molta memoria la dimensione è determinante: un array di un miliardo di `int` occupa 4 GB, uno di `double` 8 GB. Bisogna evitare l'**overflow**: un valore dell'ordine di 10^{10} non sta in un `int` e richiede un `long`. Bisogna infine conoscere la **precisione** dei reali per non incorrere in approssimazioni impreviste: un `float` conserva circa 6 cifre significative, quindi per i calcoli si preferisce il `double`.

DA RICORDARE

A cosa servono i tipi di dato? Quali sono le differenze tra i tipi e quanta memoria occupano?

Il tipo stabilisce quanta memoria occupa una variabile, come vanno interpretati i suoi bit, l'intervallo di valori rappresentabili e le operazioni consentite; permette inoltre al compilatore di scegliere le istruzioni macchina giuste (somma intera o in virgola mobile) e di segnalare usi impropri. Gli interi (char 1 byte, short 2, int 4, long 8 su Linux a 64 bit) sono rappresentati esattamente in complemento a due, o senza segno se unsigned, e rischiano solo l'overflow. I reali float (4 byte, circa 6 cifre) e double (8 byte, circa 15 cifre) seguono IEEE 754: hanno un intervallo enorme ma precisione limitata, quindi approssimano. La dimensione esatta di un tipo si ottiene con sizeof, i limiti con le costanti di limits.h e float.h.

Costanti: const e #define

Per dare un nome a un valore fisso il C offre due strade. La **direttiva** `#define MAX 100` definisce una **costante simbolica**: si scrive di solito prima del `main`, **senza tipo**, **senza** `=` e **senza punto e virgola**, e il preprocessore sostituisce testualmente ogni occorrenza di MAX con 100 prima della compilazione. Il **qualificatore** `const int MAX = 100;` definisce invece una variabile vera e propria, con un tipo, che deve essere inizializzata e non può essere modificata: il compilatore controlla il tipo e segnala ogni tentativo di assegnamento. Per convenzione i nomi delle costanti si scrivono in maiuscolo.

ATTENZIONE

Gli errori tipici con #define

Poiché `#define` fa una sostituzione testuale, `#define MAX 100;` (con il punto e virgola) inserisce `100;` ovunque compaia MAX, e `int v[MAX];` diventa `int v[100;];`, che è un errore di sintassi segnalato in un punto lontano dalla vera causa. Allo stesso modo `#define DOPPIO x+x` usato in `3 * DOPPIO` produce `3 * x + x`, non `3 * (x + x)`: le espressioni nelle macro vanno sempre racchiuse tra parentesi.

CAPITOLO 7

Operatori, espressioni e conversioni

Gli operatori aritmetici, relazionali e logici, le regole di precedenza, le conversioni di tipo implicite ed esplicite e l'idea che in C ogni comando sia la valutazione di un'espressione.

Operatori ed espressioni

OPERATORI ARITMETICI

Gli operatori aritmetici sono `+`, `-`, `*`, `/` e `%`. La loro semantica dipende dal tipo degli operandi. Se **entrambi gli operandi sono interi**, `/` esegue la **divisione intera**, che tronca verso zero la parte frazionaria: `5 / 2` vale 2, `2 / 3` vale 0, `-7 / 2` vale −3. L'operatore **modulo** `%` restituisce il resto della divisione intera, è definito **solo tra interi** e ha il segno del dividendo: `7 % 2` vale 1, `-7 % 2` vale −1. Vale sempre (a / b) · b + a % b = a. Se almeno un operando è reale, la divisione è in virgola mobile: `5.0 / 2` vale 2.5. Il resto si usa molto per i test di divisibilità: `n % 2 == 0` è vero se n è pari, `a % b == 0` se a è multiplo di b.

ASSEGNAMENTO, ASSEGNAMENTI COMPOSTI, INCREMENTO E DECREMENTO

L'**assegnamento** `=` copia il valore di destra nella variabile di sinistra e, come espressione, vale il valore assegnato; l'associatività è da destra a sinistra, per cui `x = y = z = 2;` assegna 2 a z, poi a y, poi a x. Gli **assegnamenti composti** uniscono un'operazione e un assegnamento: `h += 1` equivale a `h = h + 1`, e analogamente `-=`, `*=`, `/=`, `%=`.

Gli operatori di **incremento** `++` e **decremento** `--` aggiungono o tolgono 1 a una variabile e hanno due forme. Nella forma **prefissa** (`++x`) la variabile viene incrementata **prima** e l'espressione vale il **nuovo** valore. Nella forma **postfissa** (`x++`) l'espressione vale il **vecchio** valore e l'incremento avviene **dopo**. Usati da soli in un'istruzione (`i++;`) le due forme sono equivalenti; la differenza conta solo quando il valore dell'espressione viene usato.

ESEMPIO SVOLTO

Prefisso e postfisso

Con `int l = 1;` si eseguono `a = ++l;` e poi `b = l++;`. Nella prima istruzione l passa a 2 e a riceve il nuovo valore, 2. Nella seconda b riceve il valore attuale di l, cioè 2, e solo dopo l passa a 3. Alla fine a = 2, b = 2, l = 3.

Un'espressione come `i = i++;` o `printf("%d %d", i, i++);` modifica e legge la stessa variabile senza un ordine definito: il risultato è **non definito** dallo standard e cambia da compilatore a compilatore. Una variabile non va mai modificata più di una volta, o modificata e letta, nella stessa espressione.

OPERATORI RELAZIONALI E LOGICI

Gli **operatori relazionali** `==`, `!=`, `<`, `<=`, `>`, `>=` confrontano due valori e restituiscono un `int`: **1 se il confronto è vero, 0 se è falso**. Gli **operatori logici** combinano condizioni: `&&` (AND) è vero se entrambi gli operandi sono veri, `||` (OR) se almeno uno è vero, `!` (NOT) nega. Ricordando che per il C è falso solo lo 0, `!5` vale 0 e `!0` vale 1. L'appartenenza a un intervallo x ∈ [a, b] si scrive `a <= x && x <= b`; l'esclusione `x < a || x > b`.

`&&` e `||` hanno la **valutazione a corto circuito** (short circuit): gli operandi sono valutati da sinistra a destra e la valutazione si ferma appena il risultato è noto. In `A && B`, se A è falso B non viene valutato; in `A || B`, se A è vero B non viene valutato. Non è un dettaglio: permette di scrivere condizioni come `i < n && v[i] != 0`, dove il secondo controllo, che accede all'array, avviene solo se l'indice è valido. Lo si è visto nell'esercizio sulle celle adiacenti di una matrice, in cui `m[h][k] > 0` viene valutato solo dopo aver verificato che h e k siano dentro i limiti.

ATTENZIONE

a <= x <= b non controlla un intervallo

In C `1 < 2 < 3` vale 1, ma anche `3 > 2 > 1` vale **0**. L'espressione si valuta da sinistra: `3 > 2` vale 1, poi si confronta `1 > 1`, che è falso. Allo stesso modo `a <= x <= b` confronta con b il risultato 0 o 1 del primo confronto, non x. La forma corretta è sempre `a <= x && x <= b`. Un altro caso: con `int x = 0;` l'istruzione `if (0 && ++x)` lascia x a 0, perché per il corto circuito ++x non viene mai eseguito.

L'**operatore condizionale** (ternario) `condizione ? espr1 : espr2` vale espr1 se la condizione è vera, espr2 altrimenti: `max = (h > k) ? h : k;` è la forma compatta di un if else che assegna il massimo.

PRECEDENZA E ASSOCIATIVITÀ

Quando un'espressione contiene più operatori, la **precedenza** stabilisce quale si applica prima e l'**associatività** l'ordine tra operatori di pari precedenza. La tabella elenca gli operatori visti finora, dalla precedenza più alta alla più bassa. Nel dubbio si usano le parentesi, che rendono anche il codice più leggibile.

OPERATORI	DESCRIZIONE	ASSOCIATIVITÀ
`()  []  ->  .  x++  x--`	chiamata, indice, accesso a campi, postfissi	da sinistra
`++x  --x  +  -  !  (tipo)  sizeof  *  &`	unari: prefissi, segno, NOT, cast, dereferenziazione, indirizzo	da destra
`*  /  %`	moltiplicativi	da sinistra
`+  -`	additivi	da sinistra
`<  <=  >  >=`	relazionali	da sinistra
`==  !=`	uguaglianza	da sinistra
`&&`	AND logico	da sinistra
`||`	OR logico	da sinistra
`?:`	condizionale	da destra
`=  +=  -=  *=  /=  %=`	assegnamento	da destra

ESEMPIO SVOLTO

Valutare un'espressione

`2 + 3 * 4 % 5`: `*` e `%` hanno la stessa precedenza e associano da sinistra, quindi prima 3 · 4 = 12, poi 12 % 5 = 2, infine 2 + 2 = **4**. `10 / 3 * 3` vale **9** e non 10: la divisione intera 10 / 3 dà 3, e 3 · 3 = 9. `!x == 0` si legge `(!x) == 0`, perché `!` ha precedenza più alta di ==.

Conversioni di tipo

CONVERSIONI IMPLICITE

Quando un operatore binario riceve operandi di tipo diverso, il compilatore li porta a un **tipo comune** prima di eseguire l'operazione, secondo le **conversioni aritmetiche usuali**. Prima i tipi «piccoli» (`char`, `short`) vengono **promossi a** `int`; poi l'operando di tipo «inferiore» viene convertito in quello «superiore» secondo la gerarchia int → long → float → double → long double. Queste promozioni allargano il tipo e **non perdono informazione**. Così in `5.0 / 2` l'intero 2 diventa 2.0 e la divisione è in virgola mobile: basta che **uno** degli operandi sia reale.

Anche l'**assegnamento** converte, ma verso il tipo della variabile di sinistra, e qui l'informazione **può andare persa**. Assegnando un reale a un intero la parte frazionaria viene **troncata** (non arrotondata): `int a = 2.8965;` mette 2 in a, `int z = 8.9;` mette 8. Assegnando un valore troppo grande a un tipo più piccolo il risultato è errato. Assegnando un intero a un `bool` si ottiene 1 per qualunque valore diverso da zero.

ATTENZIONE

La divisione è decisa dagli operandi, non dalla variabile

In `float result = 5 / 2;` la variabile è un float, ma la divisione avviene **tra due interi** e vale 2; solo dopo il 2 viene convertito in 2.0. Il risultato è 2.0, non 2.5. Stesso discorso per `double d = i / 2;` con i intero. Per ottenere 2.5 bisogna che sia reale almeno un operando: `5.0 / 2`, oppure `(float) a / b` con un cast.

CONVERSIONI ESPLICITE: IL CAST

Il **cast** è un operatore unario con cui si chiede esplicitamente al compilatore di convertire un valore in un altro tipo: si scrive `(tipo) espressione`. Serve quando la conversione implicita non è quella voluta, tipicamente per forzare una divisione reale tra interi: `a / (float) b`, oppure `(double) somma / n` per calcolare una media. Il cast ha precedenza alta: `(double) somma / n` converte somma e poi divide, mentre `(double) (somma / n)` converte il risultato di una divisione già intera, e quindi già troncata.

La conversione da reale a intero tronca verso zero: `(int) 2.8` vale 2, `(int) −2.7` vale −2. Per **arrotondare all'intero più vicino** un numero positivo si somma 0.5 prima del cast: `(int) (2.8 + 0.5)` vale 3. In alternativa si usa la funzione `round` di `math.h`, che restituisce un `double`: `(int) round(3.3)` vale 3.

OVERFLOW E VALORI SPECIALI NEI PROGRAMMI

Le situazioni descritte nel capitolo 4 si osservano direttamente in C. `INT_MAX + 1` provoca un **overflow** intero: su gcc si ottiene di norma INT_MIN, ma per lo standard il comportamento non è definito; `INT_MAX + 1L` invece è corretto, perché il letterale `1L` è un `long` e la somma avviene a 64 bit. Per un `unsigned` l'aritmetica è invece ben definita e «circolare»: con `unsigned u = 0;` l'istruzione `u--` dà 4 294 967 295. In virgola mobile, `1.0 / 0` dà `inf`, `−1.0 / 0` dà `−inf`, `0.0 / 0` dà `nan`, e `math.h` definisce le costanti `INFINITY` e `NAN`. La divisione **intera** per zero, invece, non dà infinito: è un errore a tempo di esecuzione che di solito termina il programma.

DA RICORDARE

Come funzionano le conversioni di tipo in C?

Le conversioni implicite avvengono in automatico. Nelle espressioni con operandi di tipo diverso i tipi più piccoli vengono promossi (char e short a int) e l'operando di tipo inferiore viene convertito in quello superiore (int, long, float, double), senza perdita di informazione: per avere una divisione reale basta che un operando sia reale. Negli assegnamenti il valore viene convertito nel tipo della variabile, e qui si può perdere informazione: da reale a intero la parte frazionaria è troncata. Le conversioni esplicite si fanno con il cast, (tipo) espressione, per esempio (double) somma / n per una media reale tra interi; il cast verso int tronca, per arrotondare si somma 0.5 o si usa round.

L'esecuzione dei comandi come valutazione di espressioni

In C la distinzione tra «istruzioni» ed «espressioni» è più sottile che in altri linguaggi. Un'**espressione seguita da un punto e virgola** è un'istruzione (expression statement): il computer la valuta e ne scarta il valore, e l'istruzione è utile solo per i suoi **effetti collaterali** (side effect), cioè le modifiche che produce. `x = 5;` è un'espressione di assegnamento il cui valore (5) viene scartato, ma il cui effetto collaterale è modificare x; `i++;` e `printf(...)` (che restituisce il numero di caratteri stampati) funzionano allo stesso modo. Anche `3 + 4;` è un'istruzione valida, solo inutile.

Poiché anche gli assegnamenti hanno un valore, si possono annidare dentro altre espressioni:
`int y = 2 + (x = 3);` assegna 3 a x e poi 5 a y; `x = y = z = 2;` assegna lo stesso valore a tre variabili. L'uso

più utile è nelle condizioni dei cicli, dove si legge un dato e lo si controlla insieme: `while ((ch = getchar()) != EOF)`, con le parentesi indispensabili perché `!=` precede `=`.

Lo stesso principio spiega perché le **condizioni** di `if` e `while` possono essere espressioni qualunque, non solo confronti: la condizione è vera se l'espressione vale un numero diverso da zero. `if (n % 2)` è vero per n dispari, `while (n)` cicla finché n è diverso da zero, `if (p)` è vero se il puntatore p non è nullo.

DA RICORDARE

Comandi ed espressioni in C

In C quasi ogni comando è un'espressione seguita da un punto e virgola: assegnamenti, incrementi e chiamate di funzione hanno un valore, che viene scartato, e un effetto collaterale, che è il motivo per cui si scrivono. Per questo un assegnamento può comparire dentro una condizione, e per lo stesso motivo `if (x = 5)` è sempre vero: assegna 5 a x e vale 5. Una variabile non va mai modificata e letta più volte nella stessa espressione, perché l'ordine di valutazione non è definito.

CAPITOLO 8

Input e output di base

I flussi standard del C e le funzioni per leggere e scrivere dati formattati: printf, scanf, getchar e putchar.

I tre flussi standard

Un programma C comunica con l'esterno attraverso dei **flussi** (stream) di caratteri. All'avvio ne sono già aperti tre: `stdin`, lo standard input, collegato di norma alla tastiera; `stdout`, lo standard output, collegato allo schermo; `stderr`, lo standard error, anch'esso collegato allo schermo ma destinato ai **messaggi di errore**. `printf` e `scanf` lavorano su stdout e stdin; per scrivere su stderr si usa `fprintf(stderr, ...)`. La separazione è utile perché i flussi si possono **redirigere** in modo indipendente: `./prog < input.txt > output.txt` legge da un file e scrive su un altro, mentre i messaggi su stderr continuano ad apparire a schermo (o si redirigono con `2>`).

Output formattato: printf

`printf` riceve una **stringa di formato** e un elenco di valori. I caratteri normali vengono stampati così come sono; ogni **specificatore di conversione**, che inizia con `%`, viene sostituito dal valore successivo dell'elenco, formattato secondo il tipo indicato. Lo specificatore deve corrispondere al tipo del valore: stampare un `double` con `%d` produce valori senza senso. Gli specificatori principali sono `%d` (int), `%u` (unsigned), `%ld` (long), `%f` (float e double, con 6 decimali), `%e` (notazione esponenziale), `%g` (la più compatta tra %f e %e), `%c` (carattere), `%s` (stringa), `%x` e `%o` (esadecimale e ottale), `%p` (indirizzo), `%%` (il simbolo %).

Tra `%` e la lettera si possono inserire, in quest'ordine, dei **flag**, una **larghezza** minima del campo e una **precisione** preceduta dal punto. La larghezza fa occupare al valore almeno quel numero di caratteri, allineandolo a destra; il flag `-` allinea a sinistra, `0` riempie con zeri, `+` stampa il segno anche per i positivi. La precisione, per `%f`, è il numero di cifre dopo la virgola; per `%s` è il massimo numero di caratteri stampati. Le stampe a larghezza fissa servono per produrre tabelle allineate: «ogni valore occupa 7 caratteri» si traduce in `%7d`, «media su 5 caratteri di cui due decimali» in `%5.2f`.

ISTRUZIONE	OUTPUT	ISTRUZIONE	OUTPUT
`printf("[%5d]", 42)`	`[  42]`	`printf("[%.2f]", 3.14159)`	`[3.14]`
`printf("[%-5d]", 42)`	`[42  ]`	`printf("[%8.3f]", 3.14159)`	`[  3.142]`
`printf("[%05d]", 42)`	`[00042]`	`printf("[%5.2f]", 2.5)`	`[ 2.50]`
`printf("[%+d]", 42)`	`[+42]`	`printf("[%e]", 1234.5)`	`[1.234500e+03]`
`printf("[%x]", 255)`	`[ff]`	`printf("[%.3s]", "ciao")`	`[cia]`
`printf("[%c]", 65)`	`[A]`	`printf("[%-6s]", "ciao")`	`[ciao  ]`

Le **sequenze di escape** rappresentano caratteri speciali nelle stringhe: `\n` a capo, `\t` tabulazione, `\\` la barra rovesciata, `\"` le virgolette, `\'` l'apice, `\0` il carattere nullo. Per stampare una stringa seguita da un a capo c'è anche `puts`, per un singolo carattere `putchar`.

Input formattato: scanf

`scanf` legge da stdin secondo una stringa di formato e deposita i valori letti nelle variabili indicate. Poiché deve **modificare** quelle variabili, riceve il loro **indirizzo**, ottenuto con l'operatore `&`: `scanf("%d", &x);`. Dimenticare la `&` è uno degli errori più comuni: il programma compila (con un warning) ma scrive in un indirizzo casuale. Fanno eccezione le stringhe, perché il nome di un array è già un indirizzo: `scanf("%s", nome);`. Per i `double` lo specificatore di `scanf` è `%lf`, mentre in `printf` basta `%f`.

Il funzionamento di `scanf` va conosciuto nei dettagli. Gli specificatori numerici e `%s` **saltano gli spazi bianchi** iniziali (spazi, tabulazioni, a capo), poi consumano i caratteri finché sono compatibili con il formato; `%s` si ferma al primo spazio, quindi legge una sola parola. `%c` invece **non salta gli spazi** e legge qualunque carattere, compreso l'a capo rimasto da una lettura precedente: per saltarli si scrive uno spazio nel formato, `scanf(" %c", &ch)`. `scanf` **restituisce il numero di variabili lette con successo**, oppure `EOF` se l'input è finito: controllarlo è il modo corretto di gestire input non validi. Con `%2d` si limita la lettura a 2 caratteri e con `%*d` si legge un valore senza memorizzarlo.

```
int eta;
printf("Eta': ");
if (scanf("%d", &eta) != 1) {            // nessun intero letto
    fprintf(stderr, "Input non valido\n");
    return EXIT_FAILURE;
}
```

getchar, putchar ed EOF

`getchar()` legge **un carattere** da stdin e lo restituisce; `putchar(c)` scrive un carattere su stdout. Quando l'input è finito `getchar` restituisce la costante `EOF` (end of file, di solito −1). Da tastiera la fine dell'input si segnala con Ctrl+D su Linux e macOS e Ctrl+Z su Windows. Il risultato di `getchar` va memorizzato in un `int` e non in un `char`, perché `EOF` deve essere distinguibile da tutti i 256 caratteri possibili.

```
int ch;                                  // int, non char: deve contenere anche EOF
int caratteri = 0, linee = 0;
while ((ch = getchar()) != EOF) {        // leggi finche' l'input non finisce
    caratteri++;
    if (ch == '\n')                      // un a capo chiude una linea
        linee++;
}
printf("%d linee, %d caratteri\n", linee, caratteri);
```

DA RICORDARE

Come si gestisce l'input/output in C?

L'I/O avviene tramite flussi: stdin (tastiera), stdout (schermo) e stderr (schermo, per i messaggi di errore), già aperti all'avvio e dichiarati in stdio.h. printf scrive su stdout secondo una stringa di formato con specificatori (%d, %f, %c, %s…), che possono indicare larghezza del campo, precisione e allineamento. scanf legge da stdin e riceve gli indirizzi delle variabili (operatore &), salta gli spazi per i formati numerici e restituisce il numero di valori letti, da controllare per gestire input errati. Per i singoli caratteri ci sono getchar e putchar, con EOF che segnala la fine dell'input; per gli errori si usa fprintf(stderr, …). Gli stessi meccanismi si estendono ai file con fprintf, fscanf, fgets.

CAPITOLO 9

Il controllo del flusso

Le istruzioni di selezione e di iterazione del C, le istruzioni break e continue e gli schemi ricorrenti con cui si costruiscono i cicli.

Le strutture di controllo del C realizzano i tre costrutti di Böhm e Jacopini. La **sequenza** è un blocco di istruzioni tra graffe; la **selezione** è data da `if`, `if else` e `switch`; l'**iterazione** da `while`, `do while` e `for`. Il corpo di ogni struttura è una singola istruzione oppure un **blocco** `{ … }`: conviene usare sempre le graffe, anche per una sola istruzione, per evitare errori quando se ne aggiungono altre.

if e if else

`if (condizione) istruzione` esegue l'istruzione solo se la condizione è vera;
`if (condizione) istr1 else istr2` sceglie tra due alternative. Le selezioni si possono mettere in **sequenza** (più if indipendenti, valutati tutti), **annidare** (un if dentro un ramo di un altro if) oppure concatenare nella forma **else if**, che esprime una scelta tra più alternative mutuamente esclusive: le condizioni vengono provate nell'ordine e si esegue solo il ramo della prima vera.

```
if (voto >= 90)                  // le condizioni sono provate in ordine:
    printf("A\n");               // si esegue solo il primo ramo vero
else if (voto >= 80)
    printf("B\n");               // qui si arriva solo se voto < 90
else if (voto >= 70)
    printf("C\n");
else
    printf("insufficiente\n");   // nessuna condizione vera
```

ATTENZIONE

else pendente e punto e virgola di troppo

Un `else` si associa sempre all'**if più vicino** che ne è privo, indipendentemente dall'indentazione. In `if (a > 0) if (b > 0) printf("x"); else printf("y");` l'else appartiene a `if (b > 0)`, anche se fosse scritto allineato al primo if. Per associarlo al primo bisogna racchiudere l'if interno tra graffe.

Un punto e virgola subito dopo la condizione, `if (x > 0);`, termina l'if con un'**istruzione vuota**: il blocco che segue viene eseguito sempre. Lo stesso errore dopo un `while (cond);` produce un ciclo con corpo vuoto, spesso infinito, e dopo un `for (...);` esegue il corpo una sola volta, a ciclo finito.

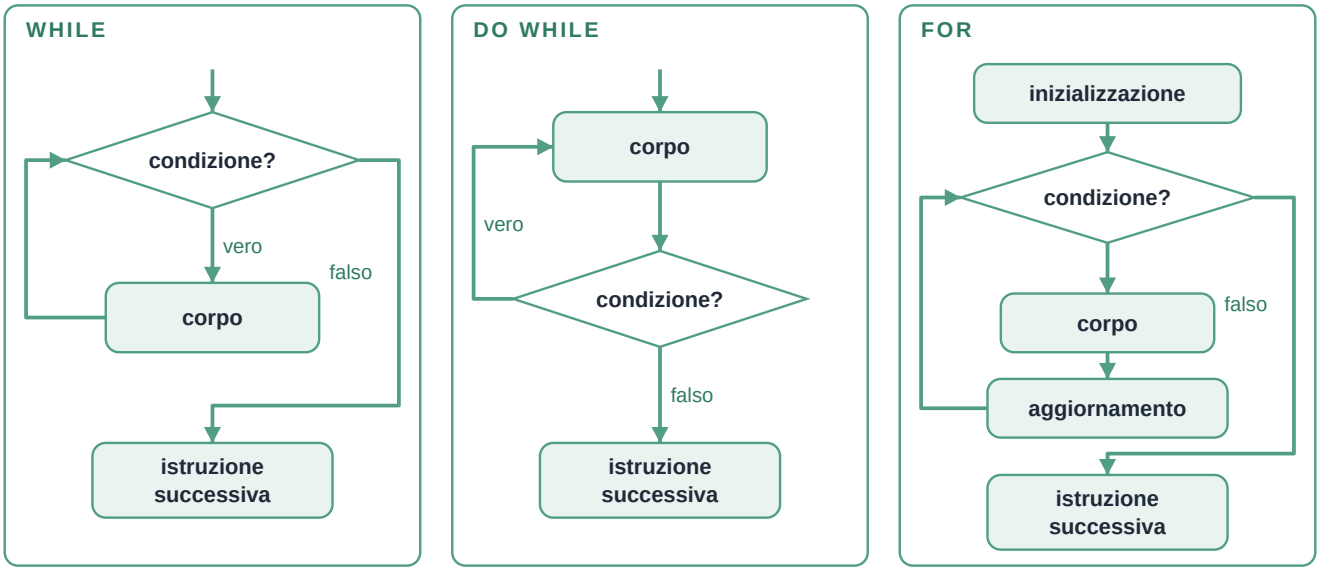
switch

Lo `switch` sceglie tra molte alternative in base al valore di un'**espressione intera** (o di un carattere, che è un intero). Ogni `case` è seguito da una **costante** intera; l'esecuzione salta al case il cui valore coincide con quello dell'espressione e prosegue da lì. L'istruzione `break` fa uscire dallo switch: **senza break l'esecuzione «cade» nel case successivo** (fall through), comportamento che si sfrutta deliberatamente per raggruppare più valori sotto le stesse istruzioni. Il ramo `default`, facoltativo, si esegue se nessun case corrisponde.

```
switch (voto) {           // voto e' un char
    case 'A':
    case 'a':             // nessun break: 'A' e 'a' eseguono lo stesso codice
        contaA++;
        break;            // esce dallo switch
    case 'B':
    case 'b':
        contaB++;
        break;
    case '\n':           // caratteri da ignorare
    case ' ':
        break;
    default:              // qualunque altro valore
        printf("Voto non valido\n");
        break;
}
```

Lo switch ha dei limiti precisi: confronta solo per **uguaglianza** con **costanti intere**; non può usare intervalli, reali o stringhe, né variabili nei case. Per questi casi si usa una catena di else if. In compenso, con molti valori discreti (un menu di scelte, un carattere) è più leggibile.

I cicli while, do while e for



I tre cicli del C

Il `while (condizione) corpo` valuta la condizione **prima** di ogni iterazione: se è falsa già la prima volta il corpo non viene mai eseguito, quindi il corpo viene eseguito **zero o più volte**. Il `do corpo while (condizione);` valuta la condizione **dopo** il corpo, che quindi viene eseguito **almeno una volta**: è il ciclo naturale per la **validazione dell'input**, in cui si chiede un valore e lo si richiede finché non è valido. Si noti il punto e virgola finale, obbligatorio nel do while.

Il `for (inizializzazione; condizione; aggiornamento) corpo` riunisce in un'unica riga le quattro parti di ogni ciclo: inizializzazione delle variabili, condizione di ripetizione, corpo, aggiornamento. È **equivalente** al while `inizializzazione; while (condizione) { corpo; aggiornamento; }` ed è la scelta naturale quando il **numero di iterazioni è noto** a priori, come nello scorrere un array. Ciascuna delle tre parti può mancare: `for (;;)` è un ciclo infinito, come `while (1)`. Dal C99 la variabile di controllo si può dichiarare nell'inizializzazione, `for (int i = 0; i < n; i++)`, e in quel caso esiste solo dentro il ciclo.

```
for (i = 0; i < N; i++)        // da 0 a N-1: N iterazioni (indici di un array)
for (i = 1; i <= N; i++)       // da 1 a N: N iterazioni
for (i = N - 1; i >= 0; i--)   // da N-1 a 0, all'indietro

do {                           // validazione: il corpo va eseguito almeno una volta
    printf("Numero tra 1 e 10: ");
    scanf("%d", &n);
} while (n < 1 || n > 10);     // ripeti finche' il valore non e' valido
```

break e continue

Dentro un ciclo, `break` termina **immediatamente** il ciclo più interno e prosegue con l'istruzione successiva, anche se la condizione è ancora vera. `continue` salta il resto del corpo e passa **all'iterazione successiva**: nel while e nel do while si torna alla valutazione della condizione, nel for si esegue prima l'aggiornamento. In un ciclo `for (x = 1; x <= 10; x++)` con `if (x == 5) break;` si stampano i valori da 1 a 4; con `if (x == 5) continue;` si stampano tutti tranne il 5. Entrambi rompono la struttura a un solo ingresso e una sola uscita, quindi vanno usati con misura; un ciclo con break si può sempre riscrivere aggiungendo una variabile flag alla condizione.

Schemi ricorrenti

Molti problemi si risolvono combinando pochi schemi fissi. Un **contatore** è una variabile inizializzata a 0 e incrementata quando si verifica un evento («quanti numeri pari»). Un **accumulatore** è inizializzato all'elemento neutro dell'operazione (0 per la somma, 1 per il prodotto) e aggiornato a ogni iterazione. Per il **massimo** (o il minimo) si inizializza il candidato con il **primo valore** letto, oppure con `INT_MIN` (`INT_MAX` per il minimo), e lo si aggiorna quando si trova un valore più grande; se serve anche la **posizione** la si memorizza insieme al valore, e usando > invece di >= si conserva la **prima** occorrenza.

Un **flag** è una variabile che vale 0 o 1 e ricorda se si è verificata una condizione. Per verificare un'**esistenza** («esiste almeno un elemento negativo?») si parte da `trovato = 0` e lo si porta a 1 al primo caso favorevole; per un'**universalità** («sono tutti positivi?») si parte da `sempre = 1` e lo si porta a 0 al primo controesempio. Quando il numero di dati non è noto si usa un **ciclo a sentinella**: si legge un dato, e finché non è un valore speciale (−1, 0, EOF) lo si elabora e si legge il successivo.

```
/* media dei voti con sentinella -1: raffinamento top down
   1) inizializza  2) leggi ed elabora finche' non arriva -1  3) stampa */
int voto, somma = 0, conta = 0;          // accumulatore e contatore
int max = INT_MIN;                       // da limits.h: minore di ogni voto

printf("Voto (-1 per finire): ");
scanf("%d", &voto);                      // lettura anticipata
while (voto != -1) {                     // -1 e' la sentinella
    somma += voto;
    conta++;
    if (voto > max)                      // nuovo massimo
        max = voto;
    printf("Voto (-1 per finire): ");
    scanf("%d", &voto);                  // lettura del dato successivo
}
if (conta > 0)                           // evita la divisione per zero
    printf("Media %.2f, massimo %d\n", (double) somma / conta, max);
else
    printf("Nessun voto inserito\n");
```

Questo esempio riassume la parte: un ciclo a sentinella con lettura anticipata, un accumulatore, un contatore e un massimo, il cast per ottenere una media reale da due interi, il controllo che evita la divisione per zero e una stampa formattata con due decimali. È anche un piccolo esempio di **raffinamento top down**: prima la scaletta in tre passi del commento, poi ciascun passo tradotto in codice.

Cicli annidati

Un ciclo può contenere un altro ciclo: per ogni iterazione del ciclo esterno, quello interno viene eseguito per intero. Con due `for` da 0 a 9 il corpo interno viene eseguito 10 × 10 = 100 volte, con le coppie (i, j) che variano come le cifre di un contachilometri: j scorre tutti i valori prima che i aumenti di uno. È lo schema per stampare figure di asterischi, per le tabelline e soprattutto per scorrere le **matrici** riga per riga. Un `break` nel ciclo interno esce solo da quello interno.

DA RICORDARE

Che cos'è la programmazione strutturata? Quali sono le principali strutture di controllo del C e quando si usano?

La programmazione strutturata costruisce i programmi solo con sequenza, selezione e iterazione, ciascuna con un solo ingresso e una sola uscita, senza salti arbitrari; il teorema di Böhm e Jacopini garantisce che bastano. In C la sequenza è il blocco tra graffe. La selezione si fa con if/else, per una o due alternative o per catene di condizioni con else if, e con switch, per scegliere tra molti valori costanti interi, ricordando il break per evitare il fall through. L'iterazione si fa con while, quando il numero di ripetizioni non è noto e il corpo può non essere mai eseguito (per esempio un ciclo a sentinella), con do while, quando il corpo va eseguito almeno una volta (validazione dell'input), e con for, quando il numero di iterazioni è noto (scorrere un array). break e continue permettono rispettivamente di uscire dal ciclo e di passare all'iterazione successiva.

CAPITOLO 10

Le funzioni

Che cos'è una funzione, come si definisce, si dichiara e si chiama, la differenza tra funzioni produttive e non produttive e i due modi di passare i parametri.

Perché le funzioni

Una **funzione** è un blocco di codice con un **nome**, che riceve dei dati in ingresso (**parametri**), svolge un compito ben preciso e può restituire un **risultato**. È la traduzione in C dei **sottoproblemi** della progettazione top down: invece di scrivere un unico `main` lungo e difficile da seguire, si scompone il problema in parti, ciascuna affidata a una funzione con un nome che ne dice lo scopo. È il principio del «divide et impera».

I vantaggi sono diversi. Il programma diventa **leggibile**, perché `main` si legge come la scaletta dell'algoritmo. Il codice si **riusa**: una funzione scritta una volta si chiama da più punti, invece di ricopiare le stesse istruzioni. Si ottiene **astrazione**: chi chiama `sqrt(x)` sa cosa fa ma non ha bisogno di sapere come lo fa. Il programma è più facile da **verificare** e **correggere**, perché ogni funzione si prova da sola. È lo stesso ragionamento che fa un'azienda quando divide il lavoro tra reparti: ognuno ha un compito, riceve del materiale e restituisce un prodotto, e chi coordina non deve sapere come lavora ogni reparto.

Il C mette a disposizione molte funzioni già pronte nelle **librerie standard**. Le funzioni matematiche di `math.h` lavorano su `double`: `sqrt(x)` radice quadrata, `pow(x, y)` potenza x^y, `fabs(x)` valore assoluto di un reale, `ceil(x)` e `floor(x)` arrotondamento per eccesso e per difetto, `round(x)` all'intero più vicino, `exp(x)`, `log(x)` (logaritmo naturale), `log10(x)`, `sin(x)`, `cos(x)`, `tan(x)` (in radianti). Per il valore assoluto di un intero c'è `abs(n)` in `stdlib.h`. Con gcc su Linux i programmi che usano `math.h` si compilano aggiungendo `-lm`.

Definizione, prototipo e chiamata

LE TRE PARTI

Una funzione si **definisce** indicando il **tipo del valore restituito**, il **nome**, la lista dei **parametri formali** tra parentesi (ciascuno con il suo tipo) e il **corpo** tra graffe. L'istruzione `return espressione;` termina la funzione e restituisce il valore al chiamante, convertito nel tipo dichiarato.

```c
#include <stdio.h>

int massimo(int a, int b);            // PROTOTIPO: tipo, nome, tipi dei parametri, ';'

int main(void)
{
    int x = 4, y = 9;
    int m = massimo(x, y);            // CHIAMATA: x e y sono i parametri attuali
    printf("Massimo: %d\n", m);
    return 0;
}

int massimo(int a, int b)             // DEFINIZIONE: a e b sono i parametri formali
{
    if (a > b)
        return a;                     // return termina subito la funzione
    return b;
}
```

Il **prototipo** (o dichiarazione) è la prima riga della definizione seguita dal punto e virgola: `int massimo(int a, int b);`. I nomi dei parametri nel prototipo sono facoltativi, `int massimo(int, int);` è equivalente. Il compilatore legge il sorgente dall'alto in basso, quindi una funzione deve essere **dichiarata prima di essere usata**: per questo i prototipi si scrivono in testa al file, prima del `main`, e le definizioni possono stare dopo. Grazie al prototipo il compilatore verifica che ogni chiamata abbia il numero giusto di argomenti, di tipo compatibile, e converte gli argomenti nel tipo dei parametri: chiamando `massimo(2.7, 5)` il 2.7 viene convertito in 2. Gli `#include` servono proprio a inserire i prototipi delle funzioni di libreria.

Nella **chiamata** `massimo(x, y)` le espressioni tra parentesi sono i **parametri attuali** (o argomenti). Al momento della chiamata vengono valutati e i loro valori vengono **copiati** nei parametri formali, nell'ordine: a riceve il valore di x, b quello di y. Poi l'esecuzione passa al corpo della funzione e, al `return`, torna al punto della chiamata, dove l'espressione `massimo(x, y)` assume il valore restituito.

FUNZIONI PRODUTTIVE E NON PRODUTTIVE

Una funzione **produttiva** restituisce un valore: ha un tipo di ritorno diverso da `void` e ogni percorso del suo corpo deve terminare con un `return espressione;`. La sua chiamata è un'**espressione** e si usa dove serve un valore: in un assegnamento, in una condizione, come argomento di un'altra funzione. Una funzione **non produttiva** (una «procedura») non restituisce nulla: il tipo di ritorno è `void`, il `return` è facoltativo e, se presente, è senza valore; la sua chiamata è un'**istruzione** usata per il suo effetto, per esempio stampare. Una funzione che non riceve parametri si dichiara con `(void)`, come `int rollDice(void)`.

```c
int bisestile(int anno)               // produttiva: restituisce 1 (vero) o 0 (falso)
{
    return (anno % 4 == 0 && anno % 100 != 0) || anno % 400 == 0;
}

void stampaRiga(int n)                // non produttiva: stampa n trattini
{
    for (int i = 0; i < n; i++)
        putchar('-');
    putchar('\n');
}
```

Le funzioni che restituiscono vero o falso, come `bisestile`, si chiamano **predicati** e si usano direttamente nelle condizioni: `if (bisestile(a))`. Per convenzione restituiscono 1 o 0, oppure un `bool`.

ATTENZIONE

Il valore di ritorno va gestito correttamente

Se una funzione produttiva arriva alla graffa di chiusura senza incontrare un `return`, il valore restituito è indefinito: gcc con `–Wall` avvisa che il controllo può raggiungere la fine della funzione senza un valore. Al contrario, ignorare il valore di una funzione produttiva è lecito (si fa sempre con `printf`, che restituisce il numero di caratteri stampati), ma usare il «valore» di una funzione `void`, come in `x = stampaRiga(3);`, è un errore di compilazione.

COMPOSIZIONE DI FUNZIONI

Poiché la chiamata di una funzione produttiva è un'espressione, il suo risultato può essere passato direttamente come argomento a un'altra funzione: è la **composizione di funzioni**, come in matematica f(g(x)). `quadrato(quadrato(3))` calcola prima la chiamata interna, che vale 9, poi quella esterna, che vale 81; `printf("%d", massimo(a, massimo(b, c)))` stampa il massimo di tre valori. Composizione significa anche che una funzione può chiamarne altre: in un programma di controllo delle date, `checkdate(g, m, a)` controlla anno e mese e chiama `giorni_del_mese(m, a)`, che a sua volta chiama `bisestile(a)` per decidere se febbraio ha 28 o 29 giorni. Ogni funzione risolve un sottoproblema e si appoggia alle altre.

ESEMPIO SVOLTO

Inversione delle cifre (esercizio 5.28)

Si vuole una funzione che, dato 7631, restituisca 1367. L'idea è estrarre l'ultima cifra con `n % 10`, accodarla al risultato con `inverso * 10 + cifra` ed eliminarla da n con `n / 10`, finché n non vale 0.

```
int invertiCifre(int n)
{
    int inverso = 0;
    while (n > 0) {
        inverso = inverso * 10 + n % 10;   // accoda l'ultima cifra di n
        n = n / 10;                        // toglie l'ultima cifra da n
    }
    return inverso;
}
```

Traccia con n = 7631: inverso = 1, n = 763; inverso = 13, n = 76; inverso = 136, n = 7; inverso = 1367, n = 0. Si esce dal ciclo e si restituisce 1367.

Il passaggio dei parametri

PASSAGGIO PER VALORE

In C i parametri sono **sempre passati per valore**: il parametro formale è una **variabile locale** della funzione che riceve una **copia** del valore del parametro attuale. La funzione può modificare liberamente la sua copia, ma **la variabile del chiamante non cambia**. È come dare a qualcuno una fotocopia di un documento: può scriverci sopra quanto vuole, l'originale resta intatto. Questo protegge il chiamante da modifiche indesiderate, ma impedisce a una funzione di modificare le variabili di chi la chiama.

```
void scambiaSbagliato(int a, int b)   // a e b sono COPIE di x e y
{
    int t = a;
    a = b;
    b = t;                            // scambia le copie, non gli originali
}
/* nel main: x = 1, y = 2; scambiaSbagliato(x, y);  ->  x e y valgono ancora 1 e 2 */
```

PASSAGGIO PER RIFERIMENTO TRAMITE PUNTATORI

Quando una funzione deve modificare una variabile del chiamante, o restituire più di un risultato, si usa il **passaggio per riferimento**. In C lo si realizza passando l'**indirizzo** della variabile, ottenuto con l'operatore `&`: il parametro formale è un **puntatore** (dichiarato con `*`) e, attraverso l'operatore di dereferenziazione `*`, la funzione legge e scrive direttamente la variabile originale. Tecnicamente anche qui si passa una copia, ma è la copia di un indirizzo, e un indirizzo copiato indica sempre la stessa variabile: è come dare a qualcuno non la fotocopia ma l'indirizzo di casa dove si trova l'originale.

```
void scambia(int *pa, int *pb)       // pa e pb contengono gli INDIRIZZI di x e y
{
    int t = *pa;                     // *pa e' la variabile x del chiamante
    *pa = *pb;
    *pb = t;
}
/* nel main: x = 1, y = 2; scambia(&x, &y);  ->  x vale 2, y vale 1 */
```

È lo stesso meccanismo di `scanf("%d", &x)`: `scanf` deve scrivere in x, quindi riceve il suo indirizzo. Ed è il modo per restituire **più risultati**: una funzione che risolve un'equazione di secondo grado può restituire 1 o 0 a seconda che esistano soluzioni reali e scriverle in due variabili del chiamante, con il prototipo `int soluzioni(double a, double b, double c, double *x1, double *x2)` chiamato con `soluzioni(a, b, c, &x1, &x2)`. Gli **array** fanno caso a sé: il nome di un array è l'indirizzo del suo primo elemento, quindi passare un array significa sempre passarlo per riferimento. Puntatori e array sono l'argomento dei capitoli dal 13 al 17.

DA RICORDARE

Come avviene il passaggio dei parametri in C? Come si ottiene il passaggio per riferimento?

In C il passaggio è sempre per valore: al momento della chiamata il valore di ogni parametro attuale viene copiato nel corrispondente parametro formale, che è una variabile locale della funzione; le modifiche fatte dalla funzione non si riflettono sul chiamante. Il passaggio per riferimento si simula passando l'indirizzo della variabile (&x) a un parametro di tipo puntatore (`int *p`): dereferenziando il puntatore (`*p`) la funzione accede alla variabile originale e può modificarla. Si usa quando una funzione deve modificare variabili del chiamante, come swap o scanf, o restituire più risultati. Gli array sono sempre passati per riferimento, perché il loro nome è l'indirizzo del primo elemento.

CAPITOLO 11

Visibilità, durata e stack delle attivazioni

Dove una variabile è visibile e per quanto tempo esiste, come è organizzata la memoria di un programma e come le chiamate di funzione vengono gestite con lo stack delle attivazioni.

Visibilità e durata delle variabili

LO SCOPE

Lo **scope** (campo di visibilità) di un identificatore è la porzione di programma in cui quel nome può essere usato. Una variabile definita dentro un **blocco** `{ … }`, compreso il corpo di una funzione, è **locale** a quel blocco: è visibile dal punto della definizione fino alla graffa di chiusura e non esiste al di fuori. Anche i **parametri formali** sono locali alla funzione. Funzioni diverse possono quindi usare gli stessi nomi senza interferire: la i di una funzione e la i di un'altra sono variabili distinte. Una variabile definita **fuori da ogni funzione** è **globale**: ha scope di file, è visibile da tutte le funzioni definite dopo di essa ed esiste per tutta la durata del programma.

Se in un blocco interno si definisce una variabile con lo stesso nome di una esterna, quella interna **nasconde** l'esterna (shadowing) fino alla fine del blocco: dentro il blocco il nome si riferisce sempre alla variabile più interna. È lecito, ma rende il codice difficile da leggere.

LA DURATA: VARIABILI AUTOMATICHE E STATICHE

Lo scope dice **dove** un nome è visibile; la **durata** (classe di memorizzazione) dice **quando** la variabile esiste in memoria. Le variabili locali sono per default **automatiche**: vengono create ogni volta che l'esecuzione entra nel blocco (o la funzione viene chiamata) e distrutte quando ne esce, quindi ogni chiamata parte da zero e un'eventuale inizializzazione viene ripetuta a ogni chiamata. Una variabile locale dichiarata `static` resta visibile solo nel suo blocco, ma ha durata **statica**: viene creata e inizializzata **una sola volta**, all'avvio del programma, e **conserva il suo valore tra una chiamata e l'altra**. Le variabili globali hanno anch'esse durata statica. Le variabili statiche e globali non inizializzate valgono 0, quelle automatiche non inizializzate hanno un valore indeterminato.

```
int x = 1;                                          // globale
void useLocal(void)  { int x = 25;        printf("locale %d\n", x);  x++; }
void useStatic(void) { static int x = 50; printf("static %d\n", x);  x++; }
void useGlobal(void) {                    printf("globale %d\n", x); x *= 10; }

int main(void)
{
    int x = 5;                                      // locale a main: nasconde la globale
    { int x = 7; printf("blocco %d\n", x); }        // nasconde quella di main
    printf("main %d\n", x);
    useLocal(); useStatic(); useGlobal();
    useLocal(); useStatic(); useGlobal();
    printf("main %d\n", x);
    return 0;
}
```

ESEMPIO SVOLTO

Che cosa stampa?

L'output è: blocco 7, main 5, locale 25, static 50, globale 1, locale 25, static 51, globale 10, main 5.

Nel blocco interno x è quella definita lì (7); usciti dal blocco torna visibile la x di main (5). `useLocal` stampa sempre 25, perché la sua x automatica è ricreata e reinizializzata a ogni chiamata, e l'incremento va perso. `useStatic` stampa 50 e poi 51, perché la sua x static è inizializzata una sola volta e conserva l'incremento. `useGlobal` non ha una x propria e usa la globale: 1 alla prima chiamata, 10 alla seconda. La x di main non è mai toccata dalle altre funzioni e alla fine vale ancora 5.

Le variabili globali sembrano comode, perché ogni funzione le vede senza doverle passare come parametro, ma rendono il programma fragile: qualunque funzione può modificarle, le dipendenze tra funzioni diventano invisibili e un errore in un punto si manifesta altrove. Per questo nella buona programmazione i dati si passano attraverso parametri e valori di ritorno, e le **variabili globali** si evitano del tutto. Le costanti globali (`#define` o `const`) non pongono questo problema e si usano normalmente.

DA RICORDARE

Che cosa sono la visibilità e la durata di una variabile? Che differenza c'è tra una variabile locale, una locale static e una globale?

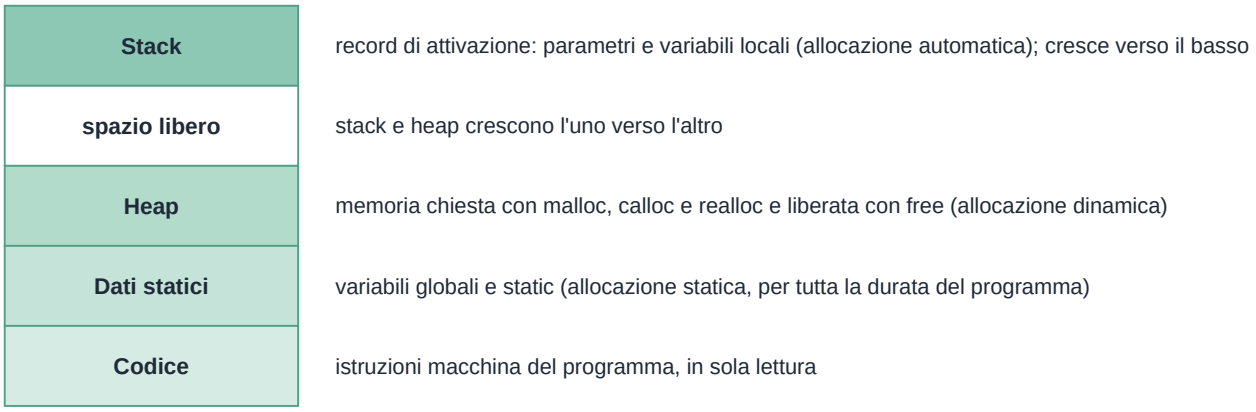
La visibilità (scope) è la parte di programma in cui un nome è utilizzabile; la durata è l'intervallo di tempo in cui la variabile esiste in memoria. Una variabile locale è visibile solo nel blocco in cui è definita ed è automatica: nasce a ogni ingresso nel blocco, sullo stack, e muore all'uscita, perdendo il valore. Una locale static ha la stessa visibilità ristretta ma durata statica: è inizializzata una volta sola e conserva il valore tra le chiamate. Una globale è definita fuori dalle funzioni, è visibile da tutte le funzioni successive del file e dura per tutto il programma. Le statiche e le globali sono allocate nell'area dati statici e valgono 0 se non inizializzate.

Lo stack delle attivazioni

LA MEMORIA DI UN PROGRAMMA

Per capire come vengono gestite le chiamate conviene guardare come il sistema operativo organizza la memoria di un programma in esecuzione. Lo spazio è diviso in zone con ruoli diversi, che corrispondono ai tre tipi di **allocazione** delle variabili. Nell'**allocazione statica** variabili globali e static vengono create all'avvio, nell'area **dati statici**, ed esistono fino alla fine. Nell'**allocazione automatica** parametri e variabili locali esistono solo per la durata della chiamata: lo spazio viene allocato sullo **stack** quando la funzione viene eseguita e liberato quando termina. L'**allocazione dinamica** non è prevista direttamente dal linguaggio, ma dalla libreria standard: le funzioni `malloc`, `calloc`, `realloc` e `free` chiedono e restituiscono memoria nello **heap**, ed è l'argomento dei capitoli dal 13 al 17.

indirizzi alti



indirizzi bassi

Organizzazione della memoria di un programma C

RECORD DI ATTIVAZIONE

Ogni volta che una funzione viene chiamata, sullo stack viene creato un **record di attivazione** (stack frame), che contiene tutto ciò che serve a quella specifica chiamata: i **parametri** con i valori ricevuti, le **variabili locali**, l'**indirizzo di ritorno** (il punto del chiamante da cui riprendere) e lo spazio per il **valore restituito**. Quando la funzione termina, il suo record viene eliminato e l'esecuzione riprende all'indirizzo di ritorno. Lo stack è una struttura **LIFO** (Last In, First Out), come una pila di piatti: si aggiunge e si toglie solo dalla cima. L'ultima funzione chiamata è sempre la prima a terminare, ed è proprio l'ordine delle chiamate annidate: se main chiama f e f chiama g, g termina prima di f e f prima di main.

Il modello spiega tutto ciò che si è visto finora. Le variabili locali sono distinte tra funzioni diverse e tra chiamate diverse della stessa funzione perché stanno in record diversi. Si perdono al termine della chiamata perché il record viene rimosso. Il passaggio per valore copia gli argomenti nel nuovo record. Restituire l'indirizzo di una variabile locale è un errore grave, perché al ritorno quella zona di stack non è più valida e verrà riusata dalla chiamata successiva. Lo stack ha una dimensione limitata (qualche megabyte): una catena di chiamate troppo profonda lo esaurisce e il programma termina con uno **stack overflow**.

FUNZIONI INLINE

Una chiamata di funzione ha un piccolo costo: creare il record di attivazione, copiare gli argomenti, saltare al codice della funzione e tornare indietro. Per funzioni brevissime, chiamate moltissime volte, questo costo può pesare più del lavoro della funzione. Il qualificatore `inline` (dal C99) suggerisce al compilatore di **sostituire la chiamata con il corpo della funzione**, evitando il salto e la creazione del record. È solo un suggerimento: il compilatore può ignorarlo, e con l'ottimizzazione attiva decide comunque da solo. Nei programmi su un unico file si scrive di solito `static inline int quadrato(int n) { return n * n; }`.

Rispetto a una macro `#define QUADRATO(n) ((n) * (n))`, che produce lo stesso effetto per sostituzione testuale, la funzione inline è più sicura: i tipi dei parametri sono controllati e gli argomenti sono valutati una sola volta. Con la macro, `QUADRATO(i++)` diventa `((i++) * (i++))` e incrementa i due volte.

CAPITOLO 12

Numeri pseudocasuali e ricorsione

La generazione di numeri pseudocasuali e le funzioni ricorsive, con il loro funzionamento in memoria e il confronto con le soluzioni iterative.

Numeri pseudocasuali

Simulazioni, giochi e test hanno bisogno di valori casuali. La funzione `rand()` di `stdlib.h` restituisce un intero compreso tra 0 e la costante `RAND_MAX` (almeno 32 767, su gcc 2 147 483 647). I numeri sono **pseudocasuali**: sono generati da un calcolo deterministico a partire da un valore iniziale, il **seme** (seed), e hanno l'aspetto di una sequenza casuale; ma a parità di seme la sequenza è **sempre la stessa**. Questo è utile durante il debug, perché permette di ripetere un'esecuzione, ma in un gioco renderebbe tutte le partite identiche.

Per cambiare sequenza a ogni esecuzione si imposta il seme con `srand(seme)`, e il seme abituale è l'ora corrente: `srand(time(NULL));`, con `time` dichiarata in `time.h`, che restituisce i secondi trascorsi dal 1° gennaio 1970. `srand` va chiamata **una sola volta**, all'inizio del main. Chiamarla dentro un ciclo è un errore classico: le chiamate ravvicinate usano lo stesso secondo come seme e producono sempre lo stesso numero.

Per ottenere un intero in un intervallo si usano **scalatura** e **traslazione**: `rand() % n` dà un valore tra 0 e n − 1 (n è il fattore di scala), e sommando a si sposta l'intervallo. La formula generale per un intero tra a e b compresi è `a + rand() % (b − a + 1)`: un dado è `1 + rand() % 6`, un numero da indovinare tra 1 e 1000 è `1 + rand() % 1000`, il lancio di una moneta `rand() % 2`. Per un reale in [0, 1) si scrive `rand() / (RAND_MAX + 1.0)`, dove il `1.0` rende reale la divisione.

```
#include <stdio.h>
#include <stdlib.h>                    // rand, srand
#include <time.h>                      // time

int lanciaDado(void)                   // restituisce un intero tra 1 e 6
{
    return 1 + rand() % 6;
}

int main(void)
{
    int frequenza[7] = {0};            // frequenza[1..6], l'indice 0 non si usa
    srand(time(NULL));                 // seme: una sola volta, all'inizio
    for (int i = 0; i < 6000; i++)
        frequenza[lanciaDado()]++;     // il risultato del lancio fa da indice
    for (int f = 1; f <= 6; f++)
        printf("%d: %d\n", f, frequenza[f]);   // circa 1000 per faccia
    return 0;
}
```

Con i dadi si può simulare il gioco del **craps**, in cui lo stato della partita (in corso, vinta, persa) è rappresentato con un tipo enumerativo `enum Stato {CONTINUA, VINTA, PERSA}` (gli `enum` sono trattati nel capitolo 18); altri usi classici sono il lancio di una moneta e il gioco «indovina il numero» tra 1 e 1000.

DA RICORDARE

Come si generano numeri casuali in C? Perché si dicono pseudocasuali?

Si usa rand() di stdlib.h, che restituisce un intero tra 0 e RAND_MAX. I numeri sono pseudocasuali perché sono prodotti da un algoritmo deterministico a partire da un seme: con lo stesso seme la sequenza si ripete identica. Per ottenere sequenze diverse a ogni esecuzione si inizializza il seme una sola volta con srand(time(NULL)). Per un intero nell'intervallo [a, b] si usa a + rand() % (b − a + 1), dove il modulo scala l'intervallo e la somma lo trasla.

La ricorsione

FUNZIONI CHE CHIAMANO SE STESSE

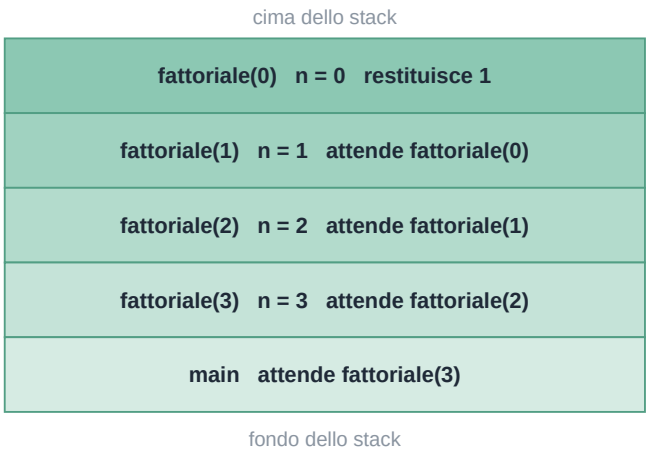
Una funzione è **ricorsiva** quando, direttamente o attraverso altre funzioni, **chiama se stessa**. La ricorsione si applica ai problemi che si possono esprimere in termini di una versione **più piccola** dello stesso problema. Ogni funzione ricorsiva ha due componenti: il **caso base**, un'istanza abbastanza semplice da essere risolta direttamente, senza ricorsione; e il **passo ricorsivo**, che riduce il problema a uno più piccolo, chiama la funzione su di esso e ne combina il risultato. Ogni chiamata deve avvicinarsi al caso base, altrimenti la ricorsione non termina.

Il fattoriale è l'esempio classico: $n! = n \cdot (n-1)!$ per $n > 0$, e $0! = 1$. La funzione ricalca esattamente la definizione matematica.

```
long fattoriale(int n)
{
    if (n <= 0)                        // caso base: 0! = 1
        return 1;
    return n * fattoriale(n - 1);      // passo ricorsivo: n! = n * (n-1)!
}
```

LO STACK DELLA RICORSIONE

Ogni chiamata ricorsiva è una chiamata come le altre, con il **proprio record di attivazione** e quindi la propria copia del parametro n. Chiamando `fattoriale(3)` dal main, la funzione non può calcolare subito 3 · fattoriale(2): deve prima chiamare `fattoriale(2)`, che chiama `fattoriale(1)`, che chiama `fattoriale(0)`. Sullo stack si accumulano quattro record, ciascuno con il suo n e ciascuno «in attesa» del risultato della chiamata successiva: è la **fase di discesa**. `fattoriale(0)` raggiunge il caso base e restituisce 1; il suo record viene rimosso e inizia la **fase di risalita**: `fattoriale(1)` riceve 1 e restituisce 1 · 1, `fattoriale(2)` restituisce 2 · 1, `fattoriale(3)` restituisce 3 · 2 = 6 al main.



Lo stack delle attivazioni nel calcolo di fattoriale(3)

Se manca il caso base, o se il passo ricorsivo non si avvicina ad esso (per esempio chiamando `fattoriale(n + 1)`), le chiamate continuano a impilare record finché lo stack si esaurisce e il programma termina con uno **stack overflow** (su Linux, «Segmentation fault»). È l'equivalente ricorsivo del ciclo infinito.

ALTRI ESEMPI

La **successione di Fibonacci** (0, 1, 1, 2, 3, 5, 8, …) è definita da fib(0) = 0, fib(1) = 1 e fib(n) = fib(n − 1) + fib(n − 2). La traduzione ricorsiva è immediata, ma ogni chiamata ne genera **due**, e gli stessi valori vengono ricalcolati moltissime volte: per fib(5) servono 15 chiamate, per fib(30) oltre 2,6 milioni. Il numero di chiamate cresce in modo esponenziale, mentre la versione iterativa con un ciclo e due variabili richiede solo n passi.

```
long fibonacci(int n)
{
    if (n == 0 || n == 1)              // due casi base
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);   // due chiamate ricorsive
}

int mcd(int a, int b)                  // Euclide: MCD(a, b) = MCD(b, a % b)
{
    if (b == 0)                        // caso base: MCD(a, 0) = a
        return a;
    return mcd(b, a % b);
}

void stampaContrario(const char *s)    // stampa una stringa al contrario
{
    if (s[0] == '\0')                  // caso base: stringa vuota
        return;
    stampaContrario(&s[1]);            // prima il resto della stringa...
    putchar(s[0]);                     // ...poi il primo carattere
}
```

La **torre di Hanoi** è il problema in cui la ricorsione mostra tutta la sua eleganza. Ci sono tre pioli e n dischi di diametro decrescente impilati sul primo; bisogna spostarli tutti sul terzo, un disco alla volta, senza mai mettere un disco sopra uno più piccolo. La soluzione ricorsiva è: per spostare n dischi da sorgente a destinazione, si spostano prima n − 1 dischi da sorgente al piolo d'appoggio, poi il disco più grande da sorgente a destinazione, infine gli n − 1 dischi dal piolo d'appoggio a destinazione. Il caso base è n = 0, in cui non c'è nulla da fare. Il numero di mosse è $2^n - 1$: 7 con tre dischi.

```
void hanoi(int n, char sorgente, char destinazione, char appoggio)
{
    if (n == 0)                                    // caso base: nessun disco
        return;
    hanoi(n - 1, sorgente, appoggio, destinazione);   // libera il disco piu' grande
    printf("%c -> %c\n", sorgente, destinazione);   // sposta il disco piu' grande
    hanoi(n - 1, appoggio, destinazione, sorgente);   // rimette sopra gli altri
}
/* hanoi(3, 'A', 'C', 'B'):  A->C  A->B  C->B  A->C  B->A  B->C  A->C */
```

RICORSIONE E ITERAZIONE

Ricorsione e iterazione sono due modi di ripetere: l'iterazione usa esplicitamente un ciclo, la ricorsione usa chiamate ripetute. Entrambe richiedono una condizione di terminazione (la condizione del ciclo, il caso base) e possono non terminare se questa non viene mai raggiunta. Ogni algoritmo ricorsivo si può riscrivere in forma iterativa e viceversa. La ricorsione rende spesso il codice **più breve e più vicino alla definizione** del problema (Hanoi, le strutture ad albero, molti algoritmi di ordinamento), ma ha un **costo**: ogni chiamata crea un record sullo stack, consuma tempo e memoria, e con molti livelli rischia lo stack overflow. Quando esiste una soluzione iterativa semplice, come per il fattoriale o Fibonacci, quella iterativa è più efficiente.

DA RICORDARE

Che cos'è la ricorsione? Come viene gestita in memoria?

Una funzione ricorsiva chiama se stessa su un'istanza più piccola del problema. Deve avere un caso base, risolto direttamente, e un passo ricorsivo che si avvicina al caso base e combina il risultato della chiamata. In memoria ogni chiamata, ricorsiva o no, crea un record di attivazione sullo stack, con i parametri, le variabili locali e l'indirizzo di ritorno: le chiamate ricorsive si impilano una sull'altra, ciascuna con le proprie variabili, finché non si raggiunge il caso base; poi i record vengono rimossi in ordine inverso e i risultati risalgono combinandosi. Senza caso base lo stack si esaurisce (stack overflow). Rispetto all'iterazione la ricorsione è spesso più chiara ma più costosa in tempo e memoria, e in casi come Fibonacci ricalcola molte volte gli stessi valori.

ESEMPIO SVOLTO

Contare le chiamate di fibonacci(5)

Detto C(n) il numero di chiamate necessarie per fib(n), vale C(0) = C(1) = 1 e C(n) = 1 + C(n − 1) + C(n − 2). Quindi C(2) = 1 + 1 + 1 = 3, C(3) = 1 + 3 + 1 = 5, C(4) = 1 + 5 + 3 = 9, C(5) = 1 + 9 + 5 = **15**. Il valore fib(3) viene calcolato due volte, fib(2) tre volte: è questo ricalcolo a rendere esponenziale la versione ricorsiva.

CAPITOLO 13

I puntatori

Perché un programma ha bisogno di gestire esplicitamente la memoria e quali strumenti offre il C per farlo: gli indirizzi e i puntatori.

Perché gestire esplicitamente la memoria

Nel capitolo 11 si è visto che la memoria di un programma è divisa in zone: le variabili globali e static stanno nell'area **dati statici** e durano quanto il programma (allocazione **statica**); parametri e variabili locali stanno sullo **stack** e durano quanto la chiamata (allocazione **automatica**). Entrambe hanno un limite: la quantità di memoria è decisa **mentre si scrive il programma**. Un array dichiarato `int v[100]` ha 100 elementi, né uno di più né uno di meno.

Molti problemi però conoscono la quantità di dati solo **durante l'esecuzione**: quante righe contiene il file da leggere, quanto è grande la matrice indicata nel file, quanti artigiani vanno inseriti nella lista. Inoltre i dati sullo stack spariscono quando la funzione che li ha creati termina, mentre a volte una funzione deve costruire una struttura che sopravviva alla chiamata. Per questi casi il C permette di **gestire esplicitamente la memoria**: il programma chiede al sistema un blocco di memoria della dimensione che serve, nel momento in cui serve, lo usa e lo restituisce quando non serve più. È l'**allocazione dinamica**, che avviene nello **heap**. Il prezzo è la responsabilità: è il programmatore a dover liberare ciò che ha allocato.

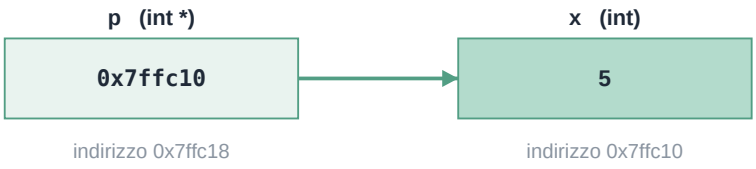
Per usare questa memoria serve un modo di riferirsi a una zona di memoria senza averle dato un nome nel sorgente: il suo **indirizzo**. Le variabili che contengono indirizzi sono i **puntatori**, che sono quindi lo strumento con cui si accede alla memoria dinamica, ma anche quello con cui si realizza il passaggio per riferimento e si lavora con gli array.

I puntatori

INDIRIZZI E PUNTATORI

Ogni variabile occupa una o più celle di memoria a partire da un certo **indirizzo**. L'**operatore di indirizzo** `&` applicato a una variabile ne restituisce l'indirizzo: `&x` è «dove si trova x». Un **puntatore** è una variabile che contiene un indirizzo; si dice che «punta» alla variabile che si trova a quell'indirizzo. Si dichiara indicando il **tipo della variabile puntata** seguito da un asterisco: `int *p;` dichiara p come puntatore a int. Il tipo è indispensabile, perché dice quanti byte leggere a partire dall'indirizzo e come interpretarli.

L'**operatore di dereferenziazione** (o di indirezione) `*`, applicato a un puntatore, dà accesso alla variabile puntata: se p contiene l'indirizzo di x, allora `*p` è x stessa, e si può sia leggere (`y = *p;`) sia scrivere (`*p = 10;`, che modifica x). Gli operatori `&` e `*` sono l'uno l'inverso dell'altro: `*&x` è x e `&*p` è p. L'analogia è quella di un indirizzo postale scritto su un foglietto: il foglietto (il puntatore) non è la casa, ma permette di raggiungerla e di cambiare ciò che c'è dentro.




```
int x = 5;
int *p = &x;   // p contiene l'indirizzo di x
*p vale 5       // il contenuto di x
*p = 10;        // modifica x
```

Un puntatore contiene l'indirizzo di un'altra variabile

ATTENZIONE

I tre significati dell'asterisco

Il simbolo `*` ha tre usi diversi a seconda del contesto. In una **dichiarazione** (`int *p;`) indica che p è un puntatore. In un'**espressione**, come operatore unario davanti a un puntatore (`*p = 10;`), è la dereferenziazione. Tra due operandi (`a * b`) è la moltiplicazione. Attenzione anche a `int *p, q;`: dichiara p puntatore e q **intero**, perché l'asterisco si lega al nome e non al tipo; per due puntatori si scrive `int *p, *q;`.

Un puntatore che non punta a nulla vale `NULL`, una costante (definita in `stdio.h` e `stdlib.h`) che corrisponde all'indirizzo 0. Si usa per inizializzare i puntatori che non hanno ancora un valore e come valore di errore: `malloc` e `fopen` restituiscono NULL se falliscono, e una lista vuota è un puntatore NULL. **Dereferenziare NULL** o un puntatore non inizializzato è un errore grave, che di solito termina il programma con un «Segmentation fault». Per questo un puntatore si inizializza sempre, a un indirizzo valido o a NULL. Gli indirizzi si stampano con `%p`.

PUNTATORI COME PARAMETRI

L'uso più semplice dei puntatori è quello già visto nel capitolo 10: il **passaggio per riferimento**. La funzione riceve l'indirizzo di una variabile del chiamante e, dereferenziandolo, la modifica. È così che funzionano `scanf`, la funzione `scambia(int *pa, int *pb)` e le funzioni che restituiscono più di un risultato, come quella che calcola il coefficiente angolare m e il termine noto q della retta per due punti,
`void retta(double x1, double y1, double x2, double y2, double *m, double *q)`, chiamata con `retta(x1, y1, x2, y2, &m, &q)`.

DA RICORDARE

A cosa servono i puntatori? Qual è la sintassi e quali sono i vantaggi rispetto alle variabili non puntatore? Come si usano per il passaggio per riferimento?

Un puntatore è una variabile che contiene l'indirizzo di un'altra variabile; si dichiara con il tipo puntato e un asterisco (`int *p`), si inizializza con l'operatore di indirizzo (`p = &x`) e si usa con l'operatore di dereferenziazione (`*p`), che dà accesso alla variabile puntata. Rispetto a una variabile normale, che contiene un valore, il puntatore permette di raggiungere e modificare dati che si trovano altrove: consente a una funzione di modificare le variabili del chiamante (passaggio per riferimento: si passa `&x` a un parametro `int *p` e si scrive `*p = …`) e di restituire più risultati; permette di passare array e strutture grandi senza copiarli; è l'unico modo di accedere alla memoria allocata dinamicamente con malloc; permette di costruire strutture dati collegate come liste, pile e code. Un puntatore che non punta a nulla vale NULL e non va dereferenziato.

CAPITOLO 14

Array e aritmetica dei puntatori

Gli array monodimensionali, il loro stretto legame con i puntatori, l'aritmetica dei puntatori e il passaggio di array alle funzioni.

Gli array

DEFINIZIONE E ACCESSO

Un **array** (vettore) è una sequenza di elementi **dello stesso tipo**, memorizzati in **celle contigue** di memoria e identificati da un unico nome. È una struttura dati **sequenziale, omogenea e contigua**. Si definisce indicando tipo, nome e numero di elementi: `int voti[30];`. Si accede al singolo elemento con l'**indice** tra parentesi quadre: `voti[0]` è il primo elemento, `voti[29]` l'ultimo. **Gli indici vanno da 0 a N − 1**: l'indice è lo spostamento dall'inizio dell'array. Ogni elemento si usa come una variabile qualunque (`voti[i] = 28;`, `somma += voti[i];`, `scanf("%d", &voti[i]);`) e l'indice può essere qualunque espressione intera.

Un array si può **inizializzare** con una lista tra graffe: `int a[5] = {10, 20, 30, 40, 50};`. Se gli inizializzatori sono meno degli elementi, i restanti valgono 0, per cui `int conta[10] = {0};` azzera tutto l'array; se si omette la dimensione, `int a[] = {1, 2, 3};`, è il compilatore a contare gli elementi. Un array locale non inizializzato contiene valori indeterminati. Il numero di elementi si ricava con `sizeof(a) / sizeof(a[0])`, ma solo dove l'array è definito: vedremo che dentro una funzione questo calcolo non funziona più.

La dimensione di un array classico deve essere una **costante**, numerica o simbolica: per questo si usa spesso `#define N 100`. Quando il numero effettivo di elementi non è noto a priori si dichiara un array di dimensione massima e si tiene in una seconda variabile l'**occupazione effettiva**: le posizioni da 0 a n − 1 sono usate, quelle da n a MAX − 1 libere, e si aggiunge in coda con `v[n] = nuovo; n++;`. Dal C99 esistono anche gli **array di lunghezza variabile** (VLA), la cui dimensione è un'espressione valutata a tempo di esecuzione: `int n; scanf("%d", &n); int v[n];`. Sono comodi, ma stanno sullo stack, non si possono inizializzare con le graffe e, se troppo grandi, esauriscono lo stack; per dimensioni grandi o dati che devono sopravvivere alla funzione si usa l'allocazione dinamica.

ATTENZIONE

Il C non controlla gli indici

Accedere a `a[5]` in un array di 5 elementi non provoca alcun errore di compilazione, e spesso nemmeno di esecuzione: il programma legge o scrive la memoria che si trova subito dopo l'array, magari un'altra variabile, con risultati imprevedibili (**buffer overflow**). L'errore tipico è il ciclo `for (i = 0; i <= N; i++)` con <= al posto di <, oppure riempire un array partendo dall'indice 1 fino a N. Tenere gli indici nell'intervallo è responsabilità del programmatore. Neppure l'assegnamento tra array esiste: `b = a;` è un errore di compilazione, e per copiare un array si copia un elemento alla volta con un ciclo.

Le operazioni elementari sugli array seguono sempre lo stesso schema, un ciclo `for (i = 0; i < n; i++)` che visita gli elementi: lettura e stampa, copia elemento per elemento, somma e media, ricerca di un valore (con un flag `trovato` e la posizione), ricerca del massimo e della sua posizione partendo da `max = v[0]`, conteggio degli elementi che soddisfano una condizione. Un uso tipico è quello dei **contatori di frequenza**: per contare quante volte esce ogni faccia di un dado si dichiara `int frequenza[7] = {0};` e si incrementa `frequenza[faccia]`, usando il risultato del lancio come indice.

DA RICORDARE

Utilità e definizione degli array. Come sono rappresentati in memoria? Come si indicizzano? Che relazione hanno con i puntatori?

Un array è una collezione di elementi dello stesso tipo identificata da un unico nome e accessibile tramite un indice: permette di gestire con un ciclo molti dati omogenei (i voti di una classe, le frequenze dei lanci) invece di dichiarare una variabile per ciascuno. Si definisce con tipo, nome e dimensione (`int v[N]`) e gli elementi sono memorizzati in celle contigue, uno dopo l'altro: l'elemento i si trova all'indirizzo iniziale più i volte la dimensione del tipo. Gli indici vanno da 0 a N − 1 e il C non controlla che siano nei limiti. Il nome dell'array, usato in un'espressione, vale l'indirizzo del suo primo elemento, quindi è un puntatore costante: `v[i]` equivale a `*(v + i)`, e un array passato a una funzione arriva come puntatore, cioè per riferimento. Le matrici sono array di array, memorizzate per righe.

Puntatori e array

IL NOME DI UN ARRAY È UN PUNTATORE

In C array e puntatori sono strettamente legati. Il **nome di un array**, usato in un'espressione, vale l'**indirizzo del suo primo elemento**: `a` equivale a `&a[0]`. Si può quindi scrivere `int *p = a;` e usare p per scorrere l'array. Il nome dell'array è però un puntatore **costante**: non si può assegnargli un altro indirizzo (`a = p;` e `a++` sono errori), mentre p è una variabile e si può spostare. Fa eccezione `sizeof(a)`, che restituisce la dimensione dell'intero array (20 byte per 5 int), mentre `sizeof(p)` restituisce la dimensione di un puntatore (8 byte su un sistema a 64 bit).

ARITMETICA DEI PUNTATORI

Sui puntatori sono definite alcune operazioni aritmetiche, che hanno senso quando il puntatore si muove all'interno di un array. Sommare un intero k a un puntatore lo sposta di **k elementi**, non di k byte: se p punta a `a[0]`, `p + 1` punta a `a[1]` e l'indirizzo cresce di `sizeof(int)`, cioè 4 byte. È per questo che il tipo del puntatore è necessario: il compilatore moltiplica k per la dimensione del tipo puntato. Si possono usare `p++`, `p--`, `p += k`. La **differenza** tra due puntatori allo stesso array dà il numero di elementi che li separa: `&a[4] − &a[0]` vale 4. Due puntatori allo stesso array si possono **confrontare** (`p < q`, `p == q`). Non ha senso invece sommare due puntatori, né fare aritmetica tra puntatori ad array diversi.

indice	a[0]	a[1]	a[2]	a[3]	a[4]
int a[5]	**10**	**20**	**30**	**40**	**50**
indirizzo	1000	1004	1008	1012	1016
puntatore	a, p	p + 1	p + 2	p + 3	p + 4
contenuto	*p	*(p+1)	*(p+2)	*(p+3)	*(p+4)

Un array in memoria: indici, indirizzi e aritmetica dei puntatori

Ne segue l'equivalenza fondamentale: `a[i]` è **per definizione** `*(a + i)`. L'indicizzazione è un modo più leggibile di scrivere «sposta di i elementi e dereferenzia». Per questo i due stili sono intercambiabili, e con un puntatore p si può scrivere sia `*(p + i)` sia `p[i]`. Si noti la differenza tra `*(p + 1)`, che è l'elemento successivo, e `*p + 1`, che è il valore dell'elemento puntato più uno: l'operatore `*` ha precedenza più alta di `+`.

```
int contaPositivi(const int *v, int n)    // v punta al primo elemento
{
    int conta = 0;
    for (int i = 0; i < n; i++)
        if (*(v + i) > 0)                   // equivale a v[i] > 0
            conta++;
    return conta;
}

int contaInversioni(const int *v, int n)  // elementi maggiori del successivo
{
    int conta = 0;
    for (const int *p = v; p < v + n - 1; p++)   // p scorre l'array
        if (*p > *(p + 1))
            conta++;
    return conta;
}
```

ESEMPIO SVOLTO

Esercizio sull'aritmetica dei puntatori

Con `int a[5] = {10, 20, 30, 40, 50}; int *p = a;` si ha: `*(p + 2)` vale 30; `p[3]` vale 40; `*p + 1` vale 11 (il valore 10 più uno, non l'elemento successivo); `&a[4] - p` vale 4. Dopo `p++`, `*p` vale 20 e `p[-1]` vale 10. Se `a` inizia all'indirizzo 1000, `p + 3` vale 1012.

Array e funzioni: dati non atomici

PASSARE UN ARRAY A UNA FUNZIONE

Un array è un dato **non atomico**, composto da molti valori, e il passaggio a una funzione segue regole proprie. Quando si scrive `f(a)` si passa il nome dell'array, cioè l'**indirizzo del primo elemento**: la funzione non riceve una copia dell'array ma un puntatore all'originale. Gli array sono quindi passati **per riferimento**: se la funzione modifica `v[i]`, modifica l'array del chiamante. È efficiente (si copia un indirizzo, non migliaia di elementi) ed è ciò che permette alle funzioni di ordinamento di ordinare l'array ricevuto. I singoli elementi, invece, sono valori atomici e passano per valore: `g(a[3])` passa una copia di `a[3]`, e per farlo modificare a `g` bisogna passare `&a[3]`.

Nel parametro formale le scritture `int v[]` e `int *v` sono **equivalenti**: in entrambi i casi `v` è un puntatore. Poiché la funzione riceve solo un indirizzo, **non conosce la dimensione dell'array**: va passata come parametro separato. Dentro la funzione `sizeof(v)` restituisce la dimensione del puntatore (8), non quella dell'array, e gcc lo segnala con un warning.

```
double media(const double v[], int n)     // n va passato: la funzione non lo conosce
{
    double somma = 0.0;
    for (int i = 0; i < n; i++)
        somma += v[i];
    return somma / n;
}

void raddoppia(int v[], int n)            // modifica l'array del chiamante
{
    for (int i = 0; i < n; i++)
        v[i] *= 2;
}
/* nel main: int a[5] = {0,1,2,3,4};  raddoppia(a, 5);  ->  a = {0,2,4,6,8} */
```

FUNZIONI CHE «RESTITUISCONO» UN ARRAY

Una funzione non può restituire un array come valore. Può restituire un **puntatore**, ma non a un array locale: `int *f(void) { int v[10]; …; return v; }` restituisce l'indirizzo di memoria sullo stack che smette di esistere al ritorno, e il chiamante si ritrova un **puntatore pendente** (dangling pointer). Le soluzioni corrette sono tre. La più comune è che il **chiamante fornisca l'array** e la funzione lo riempia: `void riempi(int v[], int n)`. In alternativa la funzione **alloca l'array dinamicamente** con malloc e restituisce il puntatore, lasciando al chiamante il compito di liberarlo con free. Infine si può restituire un array locale dichiarato `static`, che sopravvive alla chiamata, ma è condiviso tra tutte le chiamate.

ESEMPIO SVOLTO

Ricerca in un array come funzione

Una funzione di ricerca sequenziale restituisce la posizione della prima occorrenza della chiave, oppure −1 se non c'è; la condizione composta con && ferma il ciclo appena trova l'elemento, senza break.

```
int ricerca(const int v[], int n, int chiave)
{
    int i = 0;
    while (i < n && v[i] != chiave)   // il corto circuito evita di leggere v[n]
        i++;
    return (i < n) ? i : -1;          // -1 = non trovato
}
```

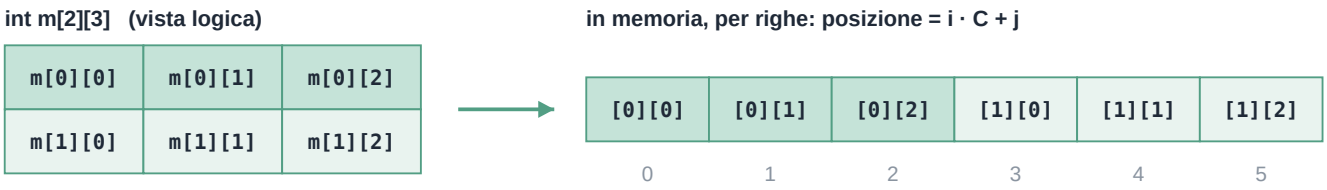
L'ordine delle due condizioni non è casuale: se fossero invertite, quando i vale n si leggerebbe `v[n]`, fuori dall'array, prima di controllare l'indice.

CAPITOLO 15

Array multidimensionali

Le matrici e gli array a più dimensioni: dichiarazione, memorizzazione per righe e uso nelle funzioni.

Un **array bidimensionale** (matrice) è un array di array: `int m[R][C];` dichiara R righe, ciascuna di C elementi, e `m[i][j]` è l'elemento di riga i e colonna j. Si inizializza riga per riga: `int m[2][3] = {{1, 2, 3}, {4, 5, 6}};`. In memoria gli elementi sono contigui e disposti **per righe** (row major): prima tutta la riga 0, poi la riga 1 e così via. L'elemento `m[i][j]` si trova quindi in posizione **i · C + j** a partire dall'inizio, ed è questa formula che il compilatore usa per calcolarne l'indirizzo. Per questo deve conoscere il numero di colonne C, mentre il numero di righe non serve.



Una matrice 2 × 3 e la sua disposizione per righe in memoria

Quando si passa una matrice a una funzione, il parametro deve specificare **tutte le dimensioni tranne la prima**: `void stampa(int m[][3], int righe)`. Con gli array di lunghezza variabile si possono passare anche le dimensioni come parametri, purché vengano **prima** della matrice: `int maxColonna(int rows, int cols, int m[][cols], int j)`. Le matrici si scorrono con **due cicli annidati**: l'esterno sulle righe e l'interno sulle colonne per una scansione per righe, il contrario per una scansione per colonne.

```
int diagonale(int n, int m[][n])          // 1 se la matrice quadrata e' diagonale
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (i != j && m[i][j] != 0)    // un elemento fuori diagonale non nullo
                return 0;
    return 1;
}

int positiviIntorno(int r, int c, int m[][c], int i, int j)
{                                          // celle adiacenti a (i,j) con valore > 0
    int conta = 0;
    for (int h = i - 1; h <= i + 1; h++)
        for (int k = j - 1; k <= j + 1; k++)
            if (h >= 0 && h < r && k >= 0 && k < c   // dentro la matrice
                && (h != i || k != j)                // esclusa la cella stessa
                && m[h][k] > 0)                      // valutato solo se indici validi
                conta++;
    return conta;
}
```

Altri schemi ricorrenti sulle matrici sono la somma di ogni riga o di ogni colonna, il massimo di una colonna, la verifica di proprietà (simmetrica: `m[i][j] == m[j][i]`; triangolare superiore: tutti gli elementi con i > j sono nulli) e l'estrazione di una sottomatrice. Per estrarre il blocco N × N nell'angolo in basso a destra di una matrice R × C, con N = min(R, C), `B[i][j]` vale `A[R-N+i][C-N+j]`.

CAPITOLO 16

Le stringhe

La rappresentazione delle stringhe come array di caratteri, la loro lettura e scrittura e le funzioni delle librerie string.h, ctype.h e stdlib.h.

Stringhe come array di caratteri

In C una **stringa** è un **array di** `char` **terminato dal carattere nullo** `'\0'`. Il terminatore è ciò che distingue una stringa da un semplice array di caratteri: tutte le funzioni sulle stringhe scorrono i caratteri finché non lo incontrano, quindi un array che deve contenere stringhe di al massimo LUN caratteri si dichiara di **LUN + 1** elementi. Si può inizializzare con una costante stringa, `char s[] = "ciao";`, che riserva 5 byte e inserisce automaticamente il terminatore, oppure carattere per carattere, `{'c', 'i', 'a', 'o', '\0'}`. Bisogna distinguere la **lunghezza** della stringa (`strlen(s)`, il numero di caratteri prima del terminatore, 4) dalla **dimensione** dell'array (`sizeof(s)`, 5): con `char t[20] = "ciao";` la lunghezza è sempre 4, la dimensione 20.

C'è differenza anche tra `char s[] = "ciao";` e `const char *s = "ciao";`. Nel primo caso s è un array modificabile, inizializzato con una copia della costante. Nel secondo s è un puntatore a una **costante stringa** memorizzata in una zona di sola lettura: modificarla (`s[0] = 'C';`) è un comportamento non definito, che di solito termina il programma. Per questo il puntatore va dichiarato `const char *`.

Per scorrere una stringa non serve conoscerne la lunghezza: basta ciclare fino al terminatore, `for (i = 0; s[i] != '\0'; i++)`. Così si contano i caratteri, si convertono in maiuscolo, si cerca un carattere. Le stringhe in quanto array si passano alle funzioni per riferimento, e la funzione non ha bisogno della dimensione perché la fine è segnata dal terminatore.

Input e output di stringhe

`printf("%s", s)` e `puts(s)` stampano una stringa (puts aggiunge un a capo). Per leggerla, `scanf("%s", s)` (senza &, perché s è già un indirizzo) legge **una sola parola**, fermandosi al primo spazio, e non controlla la dimensione dell'array: va sempre limitata, `scanf("%19s", s)` per un array di 20 caratteri. Per leggere un'**intera riga**, spazi compresi, si usa `fgets(s, dimensione, stdin)`, che legge al massimo dimensione − 1 caratteri, aggiunge il terminatore e **conserva il carattere di a capo** finale, che di solito va tolto: `s[strcspn(s, "\n")] = '\0';`. La vecchia funzione `gets` non controlla la dimensione dell'array ed è stata eliminata dallo standard C11: non va mai usata. Da una stringa già letta si estraggono valori con `sscanf`, che funziona come scanf ma legge da una stringa: `sscanf(riga, "%d%lf", &x, &y)`. La combinazione «fgets + sscanf» è il modo robusto di leggere input formattato, da tastiera o da file.

Le librerie string.h, ctype.h e le conversioni

Le stringhe non si possono assegnare né confrontare con gli operatori: `s = "ciao";` è un errore (s è un array) e `s1 == s2` confronta gli **indirizzi**, non il contenuto. Servono le funzioni di `string.h`, che presuppongono stringhe terminate correttamente e array di destinazione abbastanza grandi.

FUNZIONE	EFFETTO
strlen(s)	lunghezza di s, terminatore escluso (tipo size_t)
strcpy(dst, src)	copia src in dst, terminatore compreso; dst deve essere abbastanza grande
strncpy(dst, src, n)	copia al più n caratteri; se src è lunga n o più, NON aggiunge il terminatore

FUNZIONE	EFFETTO
strcat(dst, src)	accoda src alla fine di dst
strncat(dst, src, n)	accoda al più n caratteri di src, e aggiunge sempre il terminatore
strcmp(s1, s2)	confronto lessicografico: < 0 se s1 precede s2, 0 se uguali, > 0 se segue
strncmp(s1, s2, n)	come strcmp, ma solo sui primi n caratteri
strchr(s, c) / strrchr(s, c)	puntatore alla prima / ultima occorrenza del carattere c, o NULL
strstr(s, sub)	puntatore alla prima occorrenza della sottostringa sub, o NULL
strcspn(s, set)	lunghezza del tratto iniziale di s senza caratteri di set
strtok(s, sep)	spezza s in token separati dai caratteri di sep (modifica s)

`strcmp` confronta i caratteri uno alla volta in base al loro **codice ASCII**: `strcmp("abc", "abd")` è negativo perché 'c' precede 'd'. Poiché le maiuscole hanno codici minori delle minuscole, "Zeta" precede "alfa". È la funzione da usare per ordinare stringhe. `strtok` si chiama la prima volta con la stringa e le volte successive con NULL, e restituisce NULL quando i token sono finiti. Un caso tipico è il **controllo dell'estensione di un file**: con `strrchr(nome, '.')` si trova l'ultimo punto e con `strcmp` si confronta ciò che segue con ".txt".

```
int haEstensione(const char *nome, const char *est)   // es. haEstensione(f, ".txt")
{
    const char *punto = strrchr(nome, '.');          // ultimo '.' del nome
    return punto != NULL && strcmp(punto, est) == 0;  // NULL: nessuna estensione
}

char riga[101];                                      // 100 caratteri + '\0'
if (fgets(riga, sizeof riga, stdin) != NULL) {
    riga[strcspn(riga, "\n")] = '\0';                 // toglie l'a capo finale
    char *tok = strtok(riga, " ,");                  // primo token
    while (tok != NULL) {
        printf("[%s]\n", tok);
        tok = strtok(NULL, " ,");                     // token successivi: NULL
    }
}
```

La libreria `ctype.h` classifica e converte i singoli caratteri: `isalpha`, `isdigit`, `isalnum`, `isspace`, `isupper`, `islower`, `ispunct` restituiscono vero o falso; `toupper` e `tolower` restituiscono il carattere convertito. Le funzioni di **conversione** di `stdlib.h` trasformano stringhe in numeri: `atoi(s)` in int, `atof(s)` in double, `atol(s)` in long, e le più robuste `strtol(s, &fine, base)` e `strtod(s, &fine)`, che indicano anche dove si è fermata la conversione. `atoi("42abc")` vale 42: la conversione si ferma al primo carattere non valido. Sono le funzioni che servono per convertire gli argomenti numerici ricevuti da riga di comando, che arrivano sempre come stringhe.

DA RICORDARE

Che cosa sono le stringhe in C e a cosa servono? Quali sono le principali operazioni?

Una stringa è una sequenza di caratteri usata per rappresentare testo (nomi, parole, righe di un file). In C non è un tipo primitivo ma un array di char terminato dal carattere nullo '\0', che segna la fine: un array per stringhe di al massimo n caratteri deve averne n + 1. Si inizializza con una costante (`char s[] = "ciao"`), si stampa con %s o puts e si legge con scanf("%s"), che legge una parola, o con fgets, che legge un'intera riga. Poiché è un array, non si può assegnare né confrontare con = e ==: si usano le funzioni di string.h, come strlen (lunghezza), strcpy e strncpy (copia), strcat (concatenazione), strcmp (confronto lessicografico), strchr e strstr (ricerca), strtok (suddivisione in token). ctype.h classifica e converte i caratteri, atoi, atof e strtol convertono stringhe in numeri, sscanf estrae valori formattati da una stringa.

CAPITOLO 17

Il qualificatore const e la memoria dinamica

Il qualificatore const applicato ai puntatori, l'allocazione dinamica della memoria nello heap e gli array di puntatori e di funzioni.

Il modificatore const con i puntatori

Il qualificatore `const` applicato ai puntatori ha due significati distinti, a seconda della posizione rispetto all'asterisco. Si legge la dichiarazione da destra verso sinistra. Con `const int *p` (puntatore a int costante) è il **dato puntato** a non poter essere modificato tramite p: `*p = 5` è un errore, mentre p può essere spostato su un'altra variabile. Con `int *const p` (puntatore costante a int) è il **puntatore** a non poter cambiare: deve essere inizializzato e punterà sempre alla stessa variabile, che però si può modificare con `*p = 5`. Con `const int *const p` sono costanti entrambi. Il nome di un array si comporta come un puntatore costante.

L'uso più importante è nei **parametri delle funzioni**, secondo il **principio del minimo privilegio**: una funzione deve avere solo i permessi che le servono. Una funzione che legge un array senza modificarlo, come una stampa o una media, dichiara il parametro `const int v[]` (o `const int *v`): se per errore provasse a modificarlo il compilatore lo segnalerebbe, e chi legge il prototipo capisce subito che l'array non verrà toccato. Le funzioni di libreria seguono questa regola: `strlen(const char *s)`, `strcpy(char *dst, const char *src)`.

DA RICORDARE

A cosa serve il modificatore const? Che differenza c'è tra `const int *p` e `int *const p`?

const dichiara che un valore non deve essere modificato, e il compilatore segnala come errore ogni tentativo di farlo. Con le variabili definisce costanti tipizzate, da inizializzare nella definizione. Con i puntatori dipende dalla posizione: in `const int *p` è costante il dato puntato, che non si può modificare tramite p, mentre p può puntare altrove; in `int *const p` è costante il puntatore, che punta sempre alla stessa variabile, modificabile tramite `*p`. Nei parametri delle funzioni const realizza il principio del minimo privilegio: una funzione che riceve un array solo per leggerlo lo dichiara const, così non può modificarlo per errore.

L'allocazione dinamica della memoria

MALLOC, CALLOC, REALLOC E FREE

Le funzioni per gestire lo heap sono dichiarate in `stdlib.h`. `malloc(n_byte)` alloca un blocco di n_byte byte **non inizializzati** e ne restituisce l'indirizzo come `void *`, un puntatore generico che si assegna a un puntatore del tipo voluto; se la memoria non è disponibile restituisce **NULL**. La dimensione si calcola sempre con `sizeof`: per n interi si scrive `malloc(n * sizeof(int))`. `calloc(n, dim)` alloca n elementi di dim byte ciascuno e li **azzera**. `realloc(p, nuova_dim)` cambia la dimensione di un blocco già allocato, conservandone il contenuto, e può spostarlo altrove: restituisce il nuovo indirizzo, o NULL se fallisce lasciando intatto il blocco originale. `free(p)` restituisce al sistema il blocco puntato da p, che deve essere un indirizzo ottenuto da malloc, calloc o realloc; `free(NULL)` non fa nulla.

```c
int n;
scanf("%d", &n);
int *v = malloc(n * sizeof(int));        // n interi nello heap
if (v == NULL) {                         // controllo obbligatorio
    fprintf(stderr, "Memoria insufficiente\n");
    exit(EXIT_FAILURE);
}
for (int i = 0; i < n; i++)              // si usa come un normale array
    v[i] = i * i;

int *tmp = realloc(v, 2 * n * sizeof(int));  // raddoppia lo spazio
if (tmp == NULL) { free(v); exit(EXIT_FAILURE); }
v = tmp;                                 // solo ora si aggiorna v

free(v);                                 // restituisce la memoria
v = NULL;                                // evita un puntatore pendente
```

La memoria dinamica ha due proprietà che la distinguono dalle variabili automatiche: la **dimensione** è decisa durante l'esecuzione, e la memoria **sopravvive alla funzione** che l'ha allocata, finché non viene liberata con free. È quindi la soluzione per gli array la cui dimensione si scopre solo leggendo l'input, per le funzioni che devono restituire un array o una struttura, e per le strutture dati che crescono e si riducono nodo per nodo, come le liste.

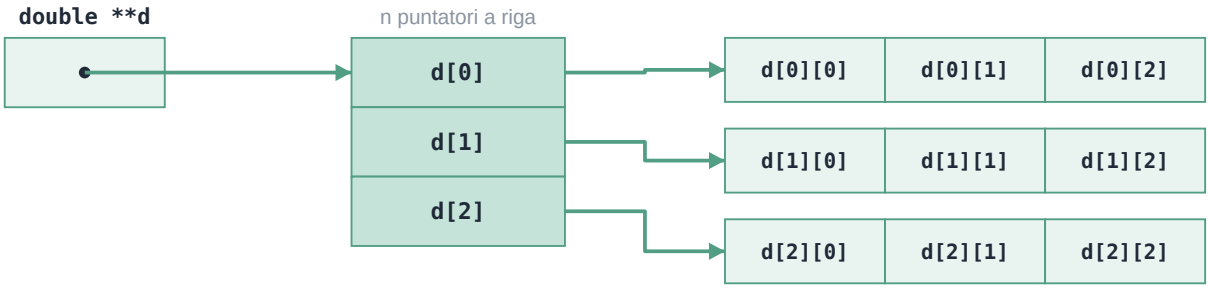
ATTENZIONE

Gli errori della memoria dinamica

Non controllare NULL: se malloc fallisce, il primo accesso dereferenzia un puntatore nullo. **Memory leak**: perdere l'unico puntatore a un blocco senza averlo liberato, per esempio riassegnando il puntatore o uscendo dalla funzione; la memoria resta occupata fino alla fine del programma. **Puntatore pendente** (dangling pointer): usare un puntatore dopo aver liberato il blocco; per questo dopo free si assegna NULL. **Doppia free**: liberare due volte lo stesso blocco. **Dimensione sbagliata**: `malloc(n)` alloca n byte, non n interi. **realloc diretto**: scrivere `v = realloc(v, …)` fa perdere il blocco originale se realloc restituisce NULL. Un programma corretto gestisce gli errori di allocazione e libera tutta la memoria prima di terminare.

MATRICI DINAMICHE

Ci sono tre modi classici di allocare una matrice N × N di double la cui dimensione N arriva da riga di comando. Nel **primo** si alloca un array di N **puntatori a riga**, `double **d = malloc(n * sizeof(double *))`, e poi ogni riga separatamente, `d[i] = malloc(n * sizeof(double))`. Si accede con la normale sintassi `d[i][j]`, perché `d[i]` è il puntatore alla riga i; ma servono N + 1 allocazioni e altrettante free, e le righe non sono contigue.



Matrice dinamica con un array di puntatori a riga

Nel **secondo** modo si alloca l'array di puntatori a riga e poi **un unico blocco** `temp` di N · N double, e si fa puntare ogni riga alla sua porzione del blocco: `d[i] = &temp[i * n]`. Si conserva la sintassi `d[i][j]`, con due sole allocazioni e dati contigui. Nel **terzo** si alloca soltanto un blocco di N · N elementi, `double *d = malloc(n * n * sizeof(double))`, e si calcola a mano la posizione dell'elemento con la formula

della memorizzazione per righe: l'elemento (i, j) è `d[i * n + j]`. È il modo più semplice da allocare e liberare (una malloc, una free). Spesso la matrice si memorizza in uno struct con righe, colonne e un puntatore `int *data`, e l'elemento (i, j) è `data[i * cols + j]`.

```
/* terzo modo: un solo blocco, indice calcolato per righe */
int *a = malloc((size_t) r * c * sizeof(int));
if (a == NULL) { fprintf(stderr, "Allocazione fallita\n"); exit(EXIT_FAILURE); }
for (int i = 0; i < r; i++)
    for (int j = 0; j < c; j++)
        a[i * c + j] = 0;                    // elemento (i, j)
free(a);

/* primo modo: array di puntatori a riga */
double **d = malloc(n * sizeof(double *));
for (int i = 0; i < n; i++)
    d[i] = malloc(n * sizeof(double));       // ogni riga a parte (controllare NULL)
/* ... uso di d[i][j] ... */
for (int i = 0; i < n; i++)
    free(d[i]);                              // prima le righe...
free(d);                                     // ...poi l'array di puntatori
```

DA RICORDARE

Come è gestita la memoria in C? Quali tipi di memoria e di allocazione esistono e con quali costrutti si accede?

La memoria di un programma è divisa in codice, dati statici, heap e stack. L'allocazione statica riguarda le variabili globali e static: sono create all'avvio nell'area dati e durano tutto il programma. L'allocazione automatica riguarda parametri e variabili locali: sono create sullo stack nel record di attivazione a ogni chiamata e distrutte al ritorno. L'allocazione dinamica avviene nello heap su richiesta esplicita, tramite le funzioni di stdlib.h: malloc alloca un blocco di byte non inizializzati, calloc un blocco azzerato, realloc ne cambia la dimensione, free lo libera; la dimensione si decide a tempo di esecuzione e il blocco vive finché non viene liberato. Alle variabili statiche e automatiche si accede per nome, alla memoria dinamica solo tramite i puntatori restituiti da malloc e calloc, con gli operatori `*` e `[]` e l'aritmetica dei puntatori. Il programmatore deve controllare che l'allocazione non abbia restituito NULL e liberare tutto ciò che alloca, per evitare memory leak e puntatori pendenti.

Array di puntatori e puntatori a funzione

Un array può contenere puntatori. L'uso più comune è l'**array di stringhe**, cioè di puntatori a char: `const char *semi[4] = {"Cuori", "Quadri", "Fiori", "Picche"};`. Ogni elemento punta a una stringa di lunghezza qualsiasi, mentre l'alternativa `char semi[4][7]` è una matrice di caratteri in cui ogni riga ha la stessa lunghezza massima. L'array `argv` di `main`, che contiene gli argomenti della riga di comando, è proprio un array di puntatori a char (capitolo 19). Un insieme di parole lette da un file si può raccogliere in un array dinamico di stringhe, `char **arr = malloc(n * sizeof(char *))`, e il bubble sort scambia i **puntatori** alle stringhe invece di copiarne i caratteri, il che è molto più efficiente.

Anche il nome di una **funzione** è un indirizzo, quello del suo codice, e si può memorizzare in un **puntatore a funzione**: `void (*f)(int) = stampa;` dichiara f come puntatore a una funzione che riceve un int e non restituisce nulla (le parentesi intorno a `*f` sono necessarie). La funzione si chiama con `f(5)`. I puntatori a funzione permettono di passare una funzione come parametro a un'altra: una funzione di ordinamento può ricevere la funzione di confronto, `void ordina(int v[], int n, int (*confronta)(int, int))`, e ordinare in modo crescente o decrescente a seconda della funzione passata. È ciò che fa la funzione di libreria `qsort`.

CAPITOLO 18

Strutture, unioni ed enumerazioni

I tipi di dato aggregati definiti dal programmatore: strutture, unioni ed enumerazioni.

Le strutture (struct)

DEFINIZIONE E USO

Un array raggruppa dati **dello stesso tipo**. Spesso però un'entità è descritta da dati **di tipo diverso**: un artigiano ha un nome e un cognome (stringhe), un'età, un numero di ordini e di pezzi (interi) e un turno (un carattere). Una **struttura** (struct), il **record** del programma, raggruppa sotto un unico nome più variabili, anche di tipo diverso, chiamate **campi** o membri. È come una scheda anagrafica: un unico foglio con caselle diverse, ciascuna con il suo nome.

```
struct artigiano {                  // definizione del tipo: nessuna memoria allocata
    char nome[11];                  // 10 caratteri + '\0'
    char cognome[11];
    int eta, ordini, pezzi;
    char turno;
};                                  // il punto e virgola finale e' obbligatorio

struct artigiano a;                 // una variabile di tipo struct artigiano
struct artigiano b = {"Julia", "Moore", 41, 48, 150, 'P'};   // inizializzazione

a.eta = 35;                         // accesso ai campi con l'operatore punto
strcpy(a.nome, "Alice");            // un campo stringa si assegna con strcpy
printf("%s %s %d\n", b.nome, b.cognome, b.eta);
```

La definizione `struct artigiano { … };` crea un **nuovo tipo**, ma non alloca memoria: la memoria viene allocata quando si definiscono variabili di quel tipo. Si accede ai campi con l'**operatore punto**: `a.eta` è il campo eta della variabile a, e si usa come una qualunque variabile int. L'inizializzazione con le graffe assegna i valori ai campi nell'ordine della definizione.

Per non ripetere ogni volta la parola `struct` si dà un nome al tipo con `typedef`, che crea un sinonimo di un tipo esistente: `typedef struct { … } Artigiano;` permette poi di scrivere semplicemente `Artigiano a;`. È la forma più usata nella pratica. `typedef` si usa anche per altri tipi: `typedef int *IntPtr;`, `typedef enum {…} Stato;`.

OPERAZIONI SULLE STRUTTURE

A differenza degli array, le strutture **si possono assegnare**: `b = a;` copia tutti i campi di a in b, compresi gli array contenuti nei campi. Una funzione può **ricevere** una struct come parametro e **restituirla** come valore: in entrambi i casi la struttura viene **copiata** (passaggio per valore), quindi la funzione lavora su una copia. Le strutture invece **non si possono confrontare** con ==: per stabilire se due struct sono uguali bisogna confrontarle campo per campo. La dimensione di una struct si ottiene con `sizeof` e può essere maggiore della somma dei campi, perché il compilatore inserisce byte di allineamento (padding): una struct con un char, un int e un double occupa 16 byte, non 13.

Quando la struct è grande, o la funzione deve modificarla, si passa un **puntatore** alla struct. Per accedere a un campo tramite un puntatore si usa l'**operatore freccia** –>: se `p` punta ad a, `p–>eta` equivale a `(*p).eta` (le parentesi sono necessarie perché il punto ha precedenza maggiore dell'asterisco). Se la funzione deve solo

leggere la struct, il parametro si dichiara `const Artigiano *p`: si evita la copia e si garantisce che la struct non venga modificata.

```
typedef struct {
    char nome[11], cognome[11];
    int eta, ordini, pezzi;
    char turno;
} Artigiano;

float media(Artigiano a)                 // per valore: riceve una copia
{
    return (float) a.pezzi / a.ordini;   // cast: altrimenti divisione intera
}

void stampa(const Artigiano *p)         // per puntatore: nessuna copia, sola lettura
{
    printf("%10s %10s %3d %5.2f\n", p->nome, p->cognome, p->eta, media(*p));
}

void invecchia(Artigiano *p)            // per puntatore: modifica l'originale
{
    p->eta++;                            // equivale a (*p).eta++
}
```

Le strutture si combinano con gli altri tipi. Un **array di struct**, `Artigiano elenco[100];`, si usa con `elenco[i].eta`. Una struct può contenere **array** (come le stringhe nome e cognome) e **altre struct** (struct annidate: `a.data.anno`). Infine una struct può contenere un **puntatore a una struct dello stesso tipo**: è una struttura **autoreferenziale**, come `typedef struct Nodo { Artigiano dati; struct Nodo *next; } Nodo;`, che è il mattone delle liste concatenate del capitolo 21. In questo caso il tipo va nominato anche dopo `struct` (Nodo), perché dentro la definizione il nome del typedef non esiste ancora.

DA RICORDARE

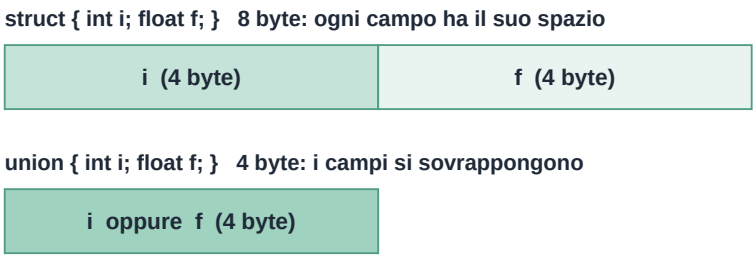
Che cos'è una struct? Come si accede ai campi e come si passa a una funzione?

Una struct è un tipo di dato aggregato definito dal programmatore che raggruppa sotto un nome più campi, anche di tipo diverso, per rappresentare un'entità (un record). Si definisce con struct nome { campi };, spesso insieme a typedef per usare il tipo senza la parola struct. Ai campi si accede con l'operatore punto (a.eta) e, tramite un puntatore, con l'operatore freccia (`p->eta`, equivalente a `(*p).eta`). Le struct si possono assegnare e restituire da una funzione (con copia di tutti i campi), ma non confrontare con ==. Passarle per valore crea una copia; per evitare la copia o per modificarle si passa un puntatore, const se la funzione deve solo leggere.

Union ed enum

LE UNION

Una **union** si definisce come una struct, ma i suoi campi **condividono la stessa area di memoria**: la union è grande quanto il suo campo più grande e in ogni momento contiene **un solo valore valido**, quello dell'ultimo campo assegnato. Se si scrive in un campo e si legge un altro, si ottiene la reinterpretazione degli stessi bit con un tipo diverso. Le union servono a risparmiare memoria quando una variabile deve contenere, in momenti diversi, valori di tipo diverso ma mai contemporaneamente; di solito si accompagnano a un campo che indica quale membro è valido in quel momento.



Una struct alloca spazio per ogni campo, una union un solo spazio condiviso

```
union numero {
    int x;
    double y;
};                                  // sizeof(union numero) = sizeof(double) = 8

union numero v;
v.x = 100;                          // ora e' valido solo v.x
v.y = 3.5;                          // stessa memoria: v.x non e' piu' valido
```

LE ENUMERAZIONI

Una **enumerazione** (enum) definisce un insieme di **costanti intere con un nome**: `enum Stato {CONTINUA, VINTA, PERSA};` crea tre costanti che valgono 0, 1 e 2. I valori si possono fissare esplicitamente, e le costanti successive proseguono da lì: in `enum Mese {GEN = 1, FEB, MAR};` FEB vale 2 e MAR 3. Un enum rende il codice più leggibile di numeri «magici»: `if (stato == VINTA)` dice molto più di `if (stato == 1)`. Nel gioco del craps lo stato della partita è un `enum Stato`; in un programma di gestione dei clienti le voci del menu sono `enum Opzioni {SALDO_ZERO = 1, SALDO_POSITIVO, SALDO_NEGATIVO, FINE};` e si usano come case di uno switch. Le costanti di un enum sono interi a tutti gli effetti, quindi si possono confrontare, usare come indici e nei case.

DA RICORDARE

Che differenza c'è tra struct e union? A cosa servono gli enum?

In una struct ogni campo ha la propria area di memoria, quindi tutti i campi contengono un valore valido contemporaneamente e la dimensione è almeno la somma dei campi. In una union tutti i campi condividono la stessa area, grande quanto il campo più grande: è valido solo l'ultimo campo assegnato, e la union si usa per risparmiare memoria quando una variabile deve contenere alternativamente valori di tipo diverso. Un enum definisce un insieme di costanti intere con nome, numerate da 0 o da valori scelti, per rendere il codice più leggibile: per esempio gli stati di una partita o le voci di un menu, da usare nei confronti e negli switch.

CAPITOLO 19

Argomenti del main e file

Come un programma riceve parametri dalla riga di comando e restituisce un codice di terminazione, e come legge e scrive file di testo e file binari.

Gli argomenti della funzione main

ARGC E ARGV

Un programma può ricevere dei parametri dalla **riga di comando** al momento dell'avvio: `./programma input.txt output.txt`. Per riceverli la funzione main si definisce con due parametri: `int main(int argc, char *argv[])`. `argc` (argument count) è il numero degli argomenti, **compreso il nome del programma**; `argv` (argument vector) è un **array di stringhe**, cioè di puntatori a char, che contiene gli argomenti: `argv[0]` è il nome del programma, `argv[1]` il primo argomento vero e proprio, e così via fino ad `argv[argc − 1]`. Con il comando dell'esempio argc vale 3, argv[1] è "input.txt" e argv[2] è "output.txt".

Gli argomenti arrivano sempre come **stringhe**: un argomento numerico va convertito, per esempio con `atoi(argv[1])` o `strtol`. Prima di usare un argomento bisogna sempre **controllare argc**: accedere ad argv[2] quando l'utente ha passato un solo argomento significa leggere oltre la fine dell'array. In caso di errore si stampa su stderr un messaggio con l'uso corretto, che contiene di solito il nome del programma `argv[0]`, e si termina con un codice d'errore.

TERMINAZIONE E CODICI DI USCITA

Il valore intero restituito da main è il **codice di terminazione** del programma, che il sistema operativo rende disponibile a chi l'ha avviato (nella shell si legge con `echo $?`). Per convenzione **0** significa che il programma è terminato correttamente, un valore **diverso da 0** che c'è stato un errore. Le costanti `EXIT_SUCCESS` ed `EXIT_FAILURE` di `stdlib.h` rendono esplicito il significato. Il programma termina quando main esegue `return`, oppure in qualunque punto con la funzione `exit(codice)`, che chiude anche i file aperti: è il modo di terminare da dentro una funzione quando si incontra un errore che non permette di proseguire. La funzione `perror("messaggio")` stampa su stderr il messaggio seguito dalla descrizione dell'ultimo errore di sistema (per esempio «No such file or directory» se fopen fallisce perché il file non esiste).

DA RICORDARE

Come si passano parametri a un programma C dalla riga di comando?

Si definisce main con due parametri, `int main(int argc, char *argv[])`. argc contiene il numero di argomenti, compreso il nome del programma, e argv è un array di stringhe: argv[0] è il nome del programma, argv[1] … argv[argc − 1] gli argomenti. Gli argomenti sono sempre stringhe, quindi quelli numerici vanno convertiti con atoi o strtol. Il programma deve controllare argc prima di accedere agli argomenti e, in caso di errore, stampare un messaggio su stderr e terminare con un codice diverso da zero (exit(EXIT_FAILURE)); il valore restituito da main è il codice di terminazione, 0 in caso di successo.

I file

FILE DI TESTO E FILE BINARI

Le variabili di un programma spariscono quando il programma termina; per conservare i dati servono i **file**, memorizzati su disco. In C un file si gestisce come un **flusso** (stream) di byte, esattamente come stdin e stdout, che sono flussi già aperti; tutte le funzioni sono in `stdio.h`. Un file aperto è rappresentato da un puntatore a una struttura di tipo `FILE`, che contiene le informazioni necessarie al sistema, tra cui la **posizione corrente**: il punto del file in cui avverrà la prossima lettura o scrittura, che avanza automaticamente a ogni operazione.

Si distinguono due tipi di file. In un **file di testo** i dati sono memorizzati come caratteri leggibili, organizzati in righe terminate da un a capo: il numero 1234 occupa quattro caratteri, '1', '2', '3', '4'. Si leggono e scrivono con le funzioni formattate (`fprintf`, `fscanf`, `fgets`, `fputs`) e si possono aprire con un editor. In un **file binario** i dati sono memorizzati **così come sono in memoria**: l'int 1234 occupa i suoi 4 byte, qualunque sia il valore. Si leggono e scrivono a blocchi con `fread` e `fwrite`; sono più compatti e più veloci, permettono di salvare intere strutture in un colpo solo e di accedere direttamente al record di posizione n, perché tutti i record hanno la stessa dimensione; non sono leggibili con un editor.

APERTURA E CHIUSURA: FOPEN E FCLOSE

`FILE *fopen(const char *nome, const char *modo)` apre il file indicato e restituisce il puntatore al FILE, oppure **NULL** se l'apertura fallisce (file inesistente in lettura, permessi insufficienti, cartella inesistente in scrittura). Il risultato va **sempre controllato**. Il **modo** stabilisce che cosa si può fare sul file.

MODO	LETTURA	SCRITTURA	IL FILE DEVE ESISTERE?	CONTENUTO ESISTENTE	POSIZIONE INIZIALE
"r"	sì	no	sì	conservato	inizio
"w"	no	sì	no (viene creato)	cancellato	inizio
"a"	no	sì, solo in coda	no (viene creato)	conservato	fine
"r+"	sì	sì	sì	conservato	inizio
"w+"	sì	sì	no (viene creato)	cancellato	inizio
"a+"	sì	sì, solo in coda	no (viene creato)	conservato	inizio per leggere, fine per scrivere

Per i file binari si aggiunge la lettera **b** al modo: `"rb"`, `"wb"`, `"ab"`, `"rb+"` (o `"r+b"`). Su Linux la b non cambia nulla, ma su Windows evita che i caratteri di a capo vengano convertiti, e va sempre messa per chiarezza. Il modo `"w"` **cancella** il contenuto di un file esistente: aprire per errore in scrittura il file di input significa perderlo.
`int fclose(FILE *f)` chiude il file: scrive su disco i dati ancora nei buffer e libera le risorse. Restituisce 0 se va tutto bene ed EOF in caso di errore. Ogni file aperto va chiuso quando non serve più.

```
FILE *f = fopen(nome, "r");
if (f == NULL) {                        // apertura fallita: messaggio ed uscita
    fprintf(stderr, "Impossibile aprire il file %s\n", nome);
    exit(EXIT_FAILURE);
}
/* ... letture ... */
fclose(f);
```

LETTURA E SCRITTURA DI FILE DI TESTO

Le funzioni sui file di testo sono le stesse di stdin e stdout, con un parametro in più che indica il file. `fprintf(f, formato, …)` scrive in modo formattato, come printf. `fscanf(f, formato, …)` legge in modo formattato, come scanf, e restituisce il numero di valori letti oppure EOF a fine file. `fgetc(f)` e `fputc(c, f)` leggono e scrivono un carattere; fgetc restituisce EOF a fine file. `fgets(s, n, f)` legge una riga (al più n − 1 caratteri, a capo compreso) e restituisce NULL a fine file; `fputs(s, f)` scrive una stringa. Con `fprintf(stderr, …)` e `fgets(s, n, stdin)` le stesse funzioni lavorano sui flussi standard.

Il punto delicato è **riconoscere la fine del file**. Il modo corretto è usare **il valore restituito dalla funzione di lettura**: si legge e, se la lettura è riuscita, si elabora il dato. La funzione `feof(f)` restituisce vero solo **dopo** che una lettura ha già tentato di andare oltre la fine, quindi non va usata come condizione del ciclo prima di leggere. I tre schemi corretti sono i seguenti.

```
/* 1) carattere per carattere */
int ch;                                          // int, per poter contenere EOF
while ((ch = fgetc(f)) != EOF) {
    /* elabora ch */
}

/* 2) riga per riga, poi estrazione dei campi con sscanf (il metodo piu' robusto) */
char riga[256];
while (fgets(riga, sizeof riga, f) != NULL) {
    char nome[11], cognome[11];
    int eta, ordini, pezzi;
    char turno;
    if (sscanf(riga, "%10s %10s %d %d %d %c",
               nome, cognome, &eta, &ordini, &pezzi, &turno) != 6) {
        fprintf(stderr, "Riga non valida\n");    // formato sbagliato
        continue;
    }
    /* elabora i campi letti */
}

/* 3) direttamente con fscanf, controllando quanti valori sono stati letti */
int conto;
char nomeCliente[30];
double saldo;
while (fscanf(f, "%d %29s %lf", &conto, nomeCliente, &saldo) == 3) {
    printf("%-10d%-13s%7.2f\n", conto, nomeCliente, saldo);
}
```

Lo schema 2 è quello consigliato: `fgets` legge sempre una riga intera, anche se è malformata, e `sscanf` permette di verificare che contenga esattamente i campi attesi; i limiti `%10s` impediscono di scrivere oltre la fine degli array. Per un file in cui i campi sono separati da virgole (formato CSV) si può spezzare la riga con `strtok(riga, ",")`. Per scrivere un file di testo si usa fprintf con le larghezze richieste dal testo: «ogni valore deve occupare 7 caratteri» si traduce in `fprintf(f, "%7d", valore)`, «media su 5 caratteri di cui 2 decimali» in `%5.2f`, e alla fine di ogni riga si scrive `"\n"`.

FILE BINARI: FREAD E FWRITE

`size_t fwrite(const void *ptr, size_t dim, size_t n, FILE *f)` scrive nel file **n elementi di dim byte** ciascuno, prelevandoli dalla memoria a partire dall'indirizzo ptr.
`size_t fread(void *ptr, size_t dim, size_t n, FILE *f)` legge dal file fino a n elementi di dim byte e li copia in memoria a partire da ptr. Entrambe **restituiscono il numero di elementi effettivamente letti o scritti**: se è minore di n la lettura è incompleta, perché il file è finito o per un errore. È questo il valore da controllare, sia per riconoscere la fine del file sia per gestire le letture incomplete.

```
int r, c;
FILE *f = fopen("dati.raw", "rb");                 // b: file binario
if (f == NULL) { perror("dati.raw"); exit(EXIT_FAILURE); }

if (fread(&r, sizeof(int), 1, f) != 1 ||           // i primi due int: righe e colonne
    fread(&c, sizeof(int), 1, f) != 1 || r <= 0 || c <= 0) {
    fprintf(stderr, "Dimensioni non valide\n");
    fclose(f);
    exit(EXIT_FAILURE);
}
int *m = malloc((size_t) r * c * sizeof(int));     // matrice r x c in un solo blocco
if (m == NULL) { fclose(f); exit(EXIT_FAILURE); }
if (fread(m, sizeof(int), (size_t) r * c, f) != (size_t) r * c) {  // r*c int
    fprintf(stderr, "Lettura incompleta\n");
    free(m);
    fclose(f);
    exit(EXIT_FAILURE);
}
fclose(f);                                          // l'elemento (i,j) e' m[i * c + j]

/* lettura di record fino alla fine del file */
struct cliente cl;
while (fread(&cl, sizeof(struct cliente), 1, g) == 1) {   // 1 = record letto
    /* elabora cl */
}
```

ACCESSO DIRETTO: FSEEK, FTELL E REWIND

Le letture e le scritture avvengono nella **posizione corrente** del file, che normalmente avanza in modo sequenziale. `int fseek(FILE *f, long offset, int origine)` sposta la posizione corrente di offset byte a partire da un'origine: `SEEK_SET` (inizio del file), `SEEK_CUR` (posizione attuale) o `SEEK_END` (fine del file); restituisce 0 se lo spostamento riesce. `long ftell(FILE *f)` restituisce la posizione corrente, in byte dall'inizio. `rewind(f)` riporta la posizione all'inizio, ed equivale a `fseek(f, 0, SEEK_SET)`; in un programma di consultazione di un archivio di clienti si usa per rileggere il file da capo a ogni richiesta dell'utente.

Con un file binario di record tutti della stessa dimensione, fseek permette l'**accesso diretto** (o casuale): il record numero k (contando da 1) inizia al byte (k − 1) · sizeof(record), quindi `fseek(f, (k − 1) * sizeof(struct cliente), SEEK_SET)` seguito da fread o fwrite legge o riscrive proprio quel record, senza leggere i precedenti. È così che si gestisce, per esempio, un archivio di 100 conti bancari in un file aperto in modo "rb+". La combinazione di fseek e ftell dà anche la **dimensione di un file**: `fseek(f, 0, SEEK_END); long byte = ftell(f);` e, per un file di interi, il numero di elementi è byte / sizeof(int).

ESEMPIO SVOLTO

fseek e ftell su un file di 5 interi

Un file binario contiene i 5 int 3, 5, 1, 2, 9 (20 byte). Dopo `fseek(f, 0, SEEK_END)`, `ftell(f)` vale 20: il file contiene 20 / 4 = 5 elementi. Dopo `fseek(f, 2 * sizeof(int), SEEK_SET)` la posizione è al byte 8, e `fread(&x, sizeof(int), 1, f)` legge il **terzo** elemento, 1; ora `ftell(f)` vale 12, perché la lettura ha fatto avanzare la posizione di 4 byte. `rewind(f)` riporta la posizione a 0, e un ciclo `while (fread(&v, sizeof(int), 1, f) == 1)` legge tutti e 5 i valori e si ferma.

DA RICORDARE

Come si gestiscono i file in C?

Un file si apre con fopen(nome, modo), che restituisce un `FILE *` da controllare perché vale NULL in caso di errore; il modo indica lettura ("r"), scrittura con cancellazione del contenuto ("w"), aggiunta in coda ("a"), le varianti con + per lettura e scrittura insieme, e la b per i file binari. Si chiude con fclose. Nei file di testo i dati sono caratteri organizzati in righe e si usano fprintf, fscanf, fgets, fputs, fgetc e fputc; nei file binari i dati sono copiati byte per byte dalla memoria e si usano fwrite e fread, che scrivono o leggono n elementi di una data dimensione e restituiscono quanti ne hanno trattati. La fine del file si riconosce dal valore restituito dalle funzioni di lettura (EOF, NULL o un numero di elementi minore del richiesto). Ogni file ha una posizione corrente: fseek la sposta (da SEEK_SET, SEEK_CUR o SEEK_END), ftell la restituisce e rewind la riporta all'inizio, permettendo l'accesso diretto ai record di un file binario.

CAPITOLO 20

Tipi di dato astratti e insiemi

Il concetto di tipo di dato astratto e la sua prima applicazione: l'insieme realizzato con un array non ordinato e con un array ordinato.

Strutture dati e tipi di dato astratti

Una **struttura dati** è un modo di organizzare i dati in memoria per poterli usare in modo efficiente. Finora si sono visti l'array, che memorizza elementi dello stesso tipo in celle contigue, e la struct, che raggruppa campi diversi. Combinando struct, puntatori e allocazione dinamica si ottengono strutture più flessibili, che **crescono e si riducono durante l'esecuzione**.

Conviene distinguere due livelli. La **descrizione formale** di una struttura, il **tipo di dato astratto**, dice **quali operazioni** offre e **come si comportano**, senza dire come sono realizzate: una pila è definita dalle operazioni push, pop e isEmpty e dalla regola che pop restituisce l'ultimo elemento inserito. L'**implementazione** stabilisce come rappresentare i dati in memoria (con un array o con una lista) e come scrivere le operazioni. La stessa struttura astratta può avere implementazioni diverse, con vantaggi e svantaggi diversi, e chi la usa non ha bisogno di conoscerle: è la stessa idea di astrazione delle funzioni, applicata ai dati. È anche il motivo per cui pila e coda si scrivono spesso in moduli separati, con l'interfaccia in un file header.

Insiemi su array

DESCRIZIONE FORMALE

Un **insieme** è una collezione di elementi **senza duplicati** e **senza un ordine** significativo. Le operazioni fondamentali sono: verificare se un elemento **appartiene** all'insieme, **inserire** un elemento (solo se non è già presente), **rimuovere** un elemento, conoscere la **cardinalità** (il numero di elementi). A partire da queste si definiscono unione, intersezione e differenza di due insiemi. Un modo semplice di implementarlo è un array di dimensione massima MAX accompagnato dal numero n di elementi presenti, riuniti in una struct.

```
#define MAX 100
typedef struct {
    int el[MAX];         // elementi: occupate le posizioni da 0 a n-1
    int n;               // cardinalita'
} Insieme;              // uso: Insieme s = {.n = 0};  (insieme vuoto)
```

ARRAY NON ORDINATO

Se gli elementi sono tenuti **in un ordine qualsiasi**, l'appartenenza richiede una **ricerca sequenziale**, che nel caso peggiore esamina tutti gli n elementi. L'inserimento, dopo il controllo dei duplicati, è immediato: il nuovo elemento va in posizione n. Anche la rimozione è semplice: poiché l'ordine non conta, il «buco» lasciato dall'elemento rimosso si riempie spostandoci l'**ultimo** elemento, senza far scorrere gli altri.

```
int appartiene(const Insieme *s, int x)
{
    for (int i = 0; i < s->n; i++)          // ricerca sequenziale
        if (s->el[i] == x)
            return 1;
    return 0;
}

int inserisci(Insieme *s, int x)            // 1 se inserito, 0 altrimenti
{
    if (s->n == MAX || appartiene(s, x))    // pieno, oppure duplicato
        return 0;
    s->el[s->n++] = x;                      // in coda: posizione n, poi n cresce
    return 1;
}

int rimuovi(Insieme *s, int x)
{
    for (int i = 0; i < s->n; i++)
        if (s->el[i] == x) {
            s->el[i] = s->el[--s->n];       // l'ultimo elemento prende il suo posto
            return 1;
        }
    return 0;
}
```

ARRAY ORDINATO

Se gli elementi sono tenuti **in ordine crescente**, la ricerca diventa molto più veloce, perché si può usare la **ricerca dicotomica** (capitolo 24), che a ogni confronto dimezza l'intervallo in cui cercare. Il prezzo si paga negli aggiornamenti: per **inserire** bisogna trovare la posizione giusta e **far scorrere di un posto a destra** tutti gli elementi maggiori, per fare spazio; per **rimuovere** bisogna far scorrere a sinistra gli elementi successivi, per chiudere il buco senza rompere l'ordine. L'array ordinato è anche più comodo per unione e intersezione, che si calcolano scorrendo i due array in parallelo come in una fusione.

```
int inserisciOrd(Insieme *s, int x)
{
    if (s->n == MAX || cercaOrd(s, x) >= 0)  // cercaOrd: ricerca dicotomica
        return 0;
    int i = s->n - 1;
    while (i >= 0 && s->el[i] > x) {        // sposta a destra i maggiori di x
        s->el[i + 1] = s->el[i];
        i--;
    }
    s->el[i + 1] = x;                       // x nel posto liberato
    s->n++;
    return 1;
}

int rimuoviOrd(Insieme *s, int x)
{
    int p = cercaOrd(s, x);                 // posizione di x, o -1
    if (p < 0)
        return 0;
    for (int i = p; i < s->n - 1; i++)       // sposta a sinistra i successivi
        s->el[i] = s->el[i + 1];
    s->n--;
    return 1;
}
```

OPERAZIONE	ARRAY NON ORDINATO	ARRAY ORDINATO
appartenenza	ricerca sequenziale: fino a n confronti	ricerca dicotomica: circa $\log_2 n$ confronti
inserimento	controllo duplicati (n confronti) + inserimento in coda immediato	ricerca + spostamento fino a n elementi
rimozione	ricerca + sostituzione con l'ultimo elemento	ricerca + spostamento fino a n elementi
stampa in ordine	richiede di ordinare prima	immediata

DA RICORDARE

Come si implementa un insieme con un array? Che differenza c'è tra array ordinato e non ordinato?

Si usa una struct con un array di dimensione massima e un contatore n degli elementi presenti; l'inserimento deve rifiutare i duplicati. Con l'array non ordinato l'appartenenza richiede una ricerca sequenziale (fino a n confronti), mentre inserimento e rimozione sono semplici: si aggiunge in posizione n e si rimuove spostando l'ultimo elemento nel buco. Con l'array ordinato l'appartenenza usa la ricerca dicotomica, molto più veloce (circa $\log_2 n$ confronti), ma inserimento e rimozione devono far scorrere gli elementi per mantenere l'ordine. La scelta dipende da quali operazioni sono più frequenti: molte ricerche favoriscono l'array ordinato, molti inserimenti quello non ordinato.

CAPITOLO 21

Liste concatenate

Le liste concatenate semplici e doppie: struttura dei nodi, operazioni fondamentali e confronto con gli array.

Liste concatenate semplici

NODI E PUNTATORI

Un array ha una dimensione fissata e inserire in mezzo costa lo spostamento di molti elementi. Una **lista concatenata** risolve entrambi i problemi: è una sequenza di **nodi** allocati dinamicamente, uno per elemento, ciascuno dei quali contiene il **dato** e un **puntatore al nodo successivo**. Il primo nodo è indicato da un puntatore detto **testa** (head); l'ultimo nodo ha il puntatore al successivo uguale a **NULL**, che segna la fine della lista; una **lista vuota** è semplicemente una testa uguale a NULL. Il nodo è una **struttura autoreferenziale**, che contiene un puntatore a una struttura del suo stesso tipo.

```
typedef struct Nodo {
    int dato;                   // il dato (anche una struct, es. Artigiano)
    struct Nodo *next;          // puntatore al nodo successivo, NULL nell'ultimo
} Nodo;

Nodo *lista = NULL;             // lista vuota
```

Rispetto all'array la lista ha vantaggi e svantaggi speculari. È **dinamica**: cresce e si riduce di un nodo alla volta, senza un limite fissato e senza sprecare spazio. **Inserire e rimuovere** un elemento, una volta trovata la posizione, richiede solo di cambiare un paio di puntatori, senza spostare nulla. Però **non c'è accesso diretto**: per raggiungere l'elemento di posizione i bisogna partire dalla testa e seguire i puntatori uno per uno, quindi niente ricerca dicotomica; e ogni nodo occupa memoria in più per il puntatore. È come la differenza tra una fila di sedie numerate (l'array: si va subito alla sedia 7, ma per inserire una persona in mezzo bisogna far spostare tutti) e una caccia al tesoro in cui ogni biglietto dice dove si trova il successivo (la lista: aggiungere una tappa è facile, ma per arrivare alla settima bisogna passare per le prime sei).

LE OPERAZIONI FONDAMENTALI

Le funzioni che **modificano la testa** (inserimento in testa, inserimento ordinato quando il nuovo elemento è il più piccolo, cancellazione del primo nodo) ricevono un **puntatore alla testa**, cioè un `Nodo **`, perché devono poter cambiare il puntatore del chiamante: è il passaggio per riferimento del capitolo 10 applicato a un puntatore. Le funzioni che **leggono** la lista (stampa, ricerca, conteggio) ricevono semplicemente la testa, `const Nodo *`. Ogni nuovo nodo si crea con malloc, controllando che non restituisca NULL.

```
void inserisciTesta(Nodo **lista, int x)       // O(1): non serve scorrere
{
    Nodo *nuovo = malloc(sizeof(Nodo));
    if (nuovo == NULL) { fprintf(stderr, "Memoria esaurita\n"); exit(EXIT_FAILURE); }
    nuovo->dato = x;
    nuovo->next = *lista;                      // il nuovo punta alla vecchia testa
    *lista = nuovo;                            // e diventa la nuova testa
}

void inserisciCoda(Nodo **lista, int x)        // bisogna arrivare all'ultimo nodo
{
    Nodo *nuovo = malloc(sizeof(Nodo));
    if (nuovo == NULL) { fprintf(stderr, "Memoria esaurita\n"); exit(EXIT_FAILURE); }
    nuovo->dato = x;
    nuovo->next = NULL;                        // sara' l'ultimo
    if (*lista == NULL) {                      // lista vuota: diventa la testa
        *lista = nuovo;
        return;
    }
    Nodo *p = *lista;
    while (p->next != NULL)                    // si ferma sull'ultimo nodo
        p = p->next;
    p->next = nuovo;
}

void stampa(const Nodo *p)
{
    for (; p != NULL; p = p->next)             // scorrimento: segui i next fino a NULL
        printf("%d -> ", p->dato);
    printf("NULL\n");
}
```

```
Nodo *cerca(Nodo *p, int x)                    // nodo che contiene x, o NULL
{
    while (p != NULL && p->dato != x)          // p->dato letto solo se p != NULL
        p = p->next;
    return p;
}

void libera(Nodo *p)                           // dealloca tutti i nodi
{
    while (p != NULL) {
        Nodo *succ = p->next;                  // salva il successivo PRIMA di free
        free(p);
        p = succ;
    }
}
```

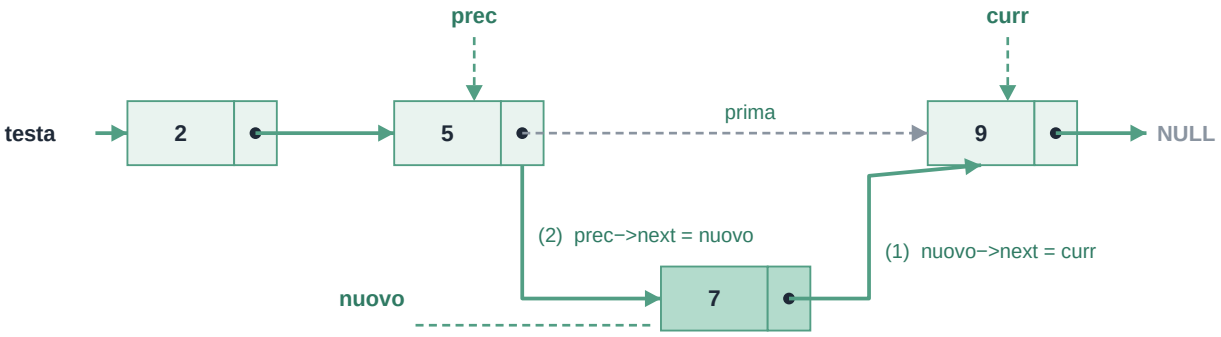
ATTENZIONE

Gli errori classici con le liste

Dereferenziare un puntatore NULL: in `while (p->dato != x && p != NULL)` l'ordine delle condizioni è sbagliato, perché alla fine della lista si legge `p->dato` con p uguale a NULL; il controllo su p va messo **prima**. Usare un nodo dopo averlo liberato: in `free(p); p = p->next;` si legge da memoria già restituita; il successivo va salvato prima. Perdere la lista: assegnare alla testa un altro valore senza aver salvato il puntatore al primo nodo rende irraggiungibili tutti i nodi (memory leak). Dimenticare il caso della lista vuota o quello in cui l'operazione riguarda il primo nodo, che richiede di modificare la testa.

INSERIMENTO IN UNA LISTA ORDINATA

Per mantenere la lista **ordinata** si cerca il punto di inserimento scorrendola con **due puntatori**: `curr` avanza sui nodi finché il loro dato è minore del nuovo valore, e `prec` resta un passo indietro, sul nodo precedente. Quando il ciclo si ferma, il nuovo nodo va inserito tra prec e curr: prima si fa puntare il nuovo nodo a curr, poi si fa puntare prec al nuovo nodo. L'ordine è importante: invertendo le due assegnazioni si perderebbe il riferimento a curr. Se prec è rimasto NULL, il nuovo valore è il più piccolo e va in testa. Lo stesso schema funziona per la lista vuota (curr è subito NULL) e per l'inserimento in fondo (curr diventa NULL).



Inserimento del valore 7 tra i nodi 5 e 9 di una lista ordinata

```
void inserisciOrdinato(Nodo **lista, int x)
{
    Nodo *nuovo = malloc(sizeof(Nodo));
    if (nuovo == NULL) { fprintf(stderr, "Memoria esaurita\n"); exit(EXIT_FAILURE); }
    nuovo->dato = x;

    Nodo *prec = NULL, *curr = *lista;
    while (curr != NULL && curr->dato < x) {   // cerca il primo nodo >= x
        prec = curr;
        curr = curr->next;
    }
    nuovo->next = curr;                        // (1) il nuovo punta a curr
    if (prec == NULL)                          // x e' il piu' piccolo: nuova testa
        *lista = nuovo;
    else                                       // (2) prec punta al nuovo
        prec->next = nuovo;
}
```

Con < nel confronto, un valore uguale a uno già presente viene inserito **prima** di esso; con <= viene inserito **dopo** gli uguali, conservando l'ordine di arrivo. Per esempio, inserendo record di artigiani in ordine di età crescente e, a parità di età, nell'ordine di lettura dal file, la condizione è `curr->dati.eta <= nuovo->dati.eta`.

CANCELLAZIONE

Per cancellare un nodo si cerca con lo stesso schema a due puntatori il nodo da eliminare; se non c'è (curr è NULL) non si fa nulla. Altrimenti lo si **scollega** facendo puntare il precedente al successivo del nodo (o spostando la testa, se il nodo è il primo) e infine lo si **libera** con free.

```
int elimina(Nodo **lista, int x)              // 1 se eliminato, 0 se assente
{
    Nodo *prec = NULL, *curr = *lista;
    while (curr != NULL && curr->dato != x) {
        prec = curr;
        curr = curr->next;
    }
    if (curr == NULL)                         // non trovato
        return 0;
    if (prec == NULL)                         // era il primo: la testa avanza
        *lista = curr->next;
    else                                      // il precedente salta curr
        prec->next = curr->next;
    free(curr);
    return 1;
}
```

SPOSTARE NODI DA UNA LISTA ALL'ALTRA

Capita di dover spostare elementi da una lista A a una lista B senza crearne copie, rimuovendo il nodo da A e inserendo lo stesso nodo in B. Si tratta di una cancellazione **senza free**: il nodo viene scollegato da A, ma resta in memoria e viene agganciato a B cambiando solo i puntatori. La funzione seguente estrae il nodo con il dato massimo e lo restituisce; il main lo inserisce in testa a B.

```
Nodo *estraiMax(Nodo **lista)                 // scollega e restituisce il nodo massimo
{
    if (*lista == NULL)
        return NULL;
    Nodo *max = *lista, *precMax = NULL;      // candidato: il primo nodo
    Nodo *prec = *lista;
    for (Nodo *p = (*lista)->next; p != NULL; prec = p, p = p->next)
        if (p->dato > max->dato) {            // > : a parita' resta il primo
            max = p;
            precMax = prec;
        }
    if (precMax == NULL)                      // il massimo era in testa
        *lista = max->next;
    else
        precMax->next = max->next;            // scollega max, senza free
    max->next = NULL;
    return max;
}

/* nel main: sposta i 3 massimi da A a B, inserendo in testa a B */
Nodo *B = NULL;
for (int i = 0; i < 3 && A != NULL; i++) {
    Nodo *n = estraiMax(&A);
    n->next = B;                              // inserimento in testa: nessuna malloc
    B = n;
}
```

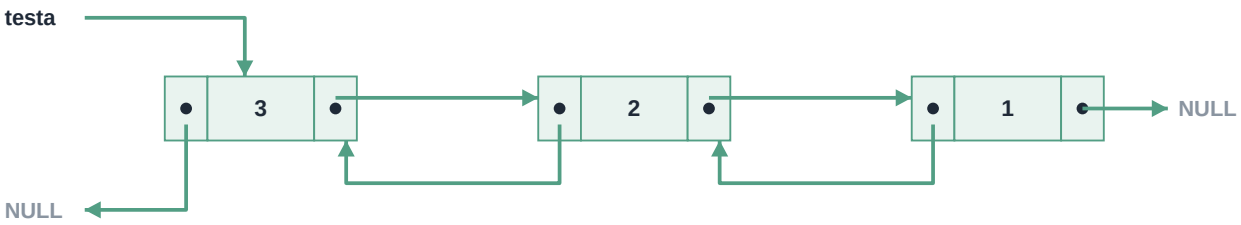
ESEMPIO SVOLTO

Spostare i due massimi

Con A = 0 → 2 → 5 → 7 → 9 → 10 → NULL, dopo `elimina(&A, 7)` ed `elimina(&A, 0)` resta A = 2 → 5 → 9 → 10. Il ciclo con due iterazioni estrae prima il 10 e lo mette in testa a B (B = 10), poi il 9 e lo mette in testa a B (B = 9 → 10). Alla fine A = 2 → 5 → NULL e B = 9 → 10 → NULL. Poiché gli inserimenti avvengono in testa, il massimo estratto per primo finisce in fondo a B, quindi B termina con l'elemento più grande.

Liste doppiamente concatenate

In una lista semplice si può procedere solo in avanti, e per cancellare un nodo bisogna conoscere il suo **precedente**, che si trova solo riscorrendo la lista dall'inizio. In una **lista doppiamente concatenata** ogni nodo contiene **due puntatori**: `next` al nodo successivo e `prev` al precedente. La testa ha `prev` uguale a NULL, l'ultimo nodo ha `next` uguale a NULL. Spesso si tiene anche un puntatore all'ultimo nodo, per inserire in fondo e scorrere la lista all'indietro.



Lista doppiamente concatenata: ogni nodo punta al precedente e al successivo

I vantaggi sono lo **scorrimento in entrambe le direzioni** e la **cancellazione di un nodo noto in un solo passo**, perché il precedente è raggiungibile direttamente da `n->prev`. Il costo è un puntatore in più per ogni nodo e operazioni più delicate: ogni inserimento e cancellazione deve aggiornare **quattro** collegamenti invece di due, e ricordarsi dei casi in cui il nodo è il primo o l'ultimo.

```
typedef struct NodoD {
    int dato;
    struct NodoD *prev;              // nodo precedente (NULL per la testa)
    struct NodoD *next;              // nodo successivo (NULL per l'ultimo)
} NodoD;

void inserisciTestaD(NodoD **lista, int x)
{
    NodoD *n = malloc(sizeof(NodoD));
    if (n == NULL) { fprintf(stderr, "Memoria esaurita\n"); exit(EXIT_FAILURE); }
    n->dato = x;
    n->prev = NULL;                  // diventa il primo
    n->next = *lista;
    if (*lista != NULL)
        (*lista)->prev = n;          // la vecchia testa ora ha un precedente
    *lista = n;
}

void eliminaNodoD(NodoD **lista, NodoD *n)     // n: nodo gia' individuato
{
    if (n->prev != NULL)
        n->prev->next = n->next;              // il precedente salta n
    else
        *lista = n->next;                     // n era la testa
    if (n->next != NULL)
        n->next->prev = n->prev;              // il successivo salta n all'indietro
    free(n);
}
```

DA RICORDARE

Che cos'è una lista concatenata? Che differenza c'è tra lista semplice e doppia, e tra lista e array?

Una lista concatenata è una sequenza di nodi allocati dinamicamente, ciascuno con un dato e un puntatore al successivo; la testa punta al primo nodo e l'ultimo ha next uguale a NULL. Nella lista doppia ogni nodo ha anche un puntatore al precedente, quindi si può scorrere in entrambe le direzioni e cancellare un nodo noto senza cercarne il precedente, al costo di un puntatore in più per nodo e di più collegamenti da aggiornare. Rispetto all'array la lista ha dimensione variabile durante l'esecuzione e inserimenti e cancellazioni che richiedono solo lo spostamento di puntatori, senza far scorrere elementi; in compenso non permette l'accesso diretto all'elemento di posizione i (bisogna scorrere dalla testa, quindi niente ricerca dicotomica) e usa memoria in più per i puntatori.

CAPITOLO 22

Pile e code

Le due strutture con accesso limitato alle estremità, la pila e la coda, con le implementazioni basate su array e su liste.

Pile

DESCRIZIONE FORMALE

Una **pila** (stack) è una sequenza in cui inserimenti e rimozioni avvengono **sempre alla stessa estremità**, la **cima** (top). Segue la politica **LIFO** (Last In, First Out): l'ultimo elemento inserito è il primo a uscire, come in una pila di piatti. Le operazioni sono **push** (inserisce un elemento in cima), **pop** (rimuove e restituisce l'elemento in cima), **top** o peek (restituisce l'elemento in cima senza rimuoverlo), **isEmpty** (vero se la pila è vuota) e, nell'implementazione con array, **isFull**. Non si può fare pop su una pila vuota: chi la usa deve controllare prima con isEmpty.

Le pile sono ovunque nell'informatica: lo **stack delle attivazioni** del capitolo 11 è una pila di record di attivazione; la funzione «annulla» di un editor è una pila di operazioni; il controllo delle parentesi bilanciate in un'espressione si fa mettendo in pila ogni parentesi aperta e togliendola a ogni parentesi chiusa; con una pila si inverte una sequenza, perché gli elementi escono in ordine opposto a quello di ingresso.

IMPLEMENTAZIONE CON ARRAY E CON LISTA

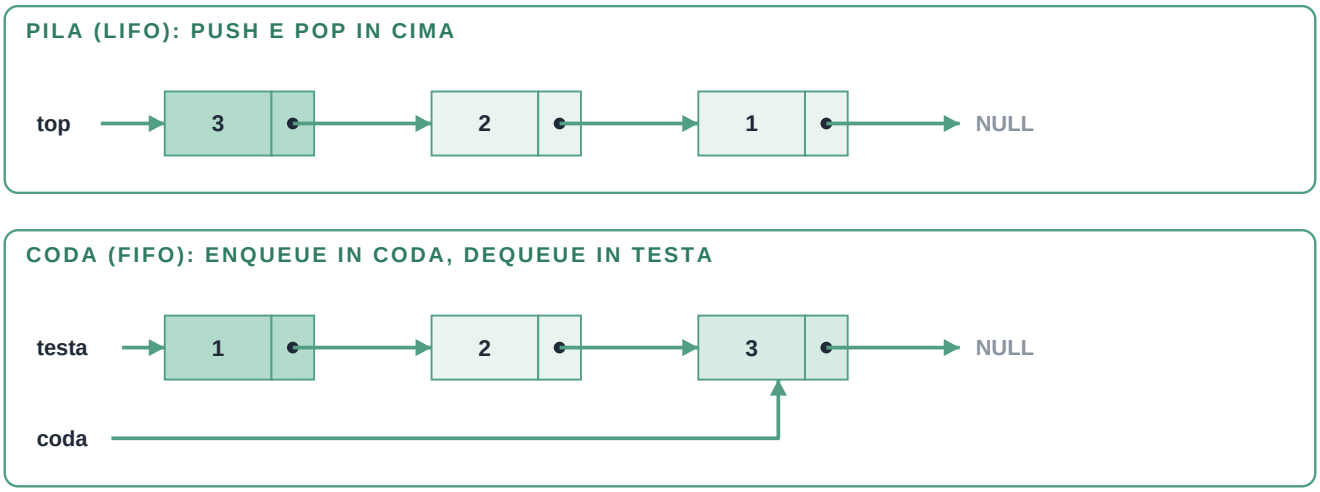
Con un **array** la pila è una struct con l'array dei dati e un indice `top`, che indica la prima posizione libera (e quindi vale anche il numero di elementi). push scrive in `dati[top]` e incrementa top; pop decrementa top e restituisce `dati[top]`. Tutte le operazioni costano un passo, ma la capacità è limitata a MAX e push deve controllare che la pila non sia piena. Con una **lista** la cima è la **testa**: push è un inserimento in testa e pop una cancellazione del primo nodo. La pila è limitata solo dalla memoria disponibile, a costo di una malloc per ogni push e una free per ogni pop.

```c
/* pila con array */
typedef struct { int dati[MAX]; int top; } PilaArray;   // top = numero di elementi
void pushA(PilaArray *p, int x) { if (p->top < MAX) p->dati[p->top++] = x; }
int  popA(PilaArray *p)         { return p->dati[--p->top]; }   // pila non vuota!
int  isEmptyA(const PilaArray *p) { return p->top == 0; }

/* pila con lista: la cima e' la testa */
void push(Nodo **top, int x)
{
    Nodo *n = malloc(sizeof(Nodo));
    if (n == NULL) { fprintf(stderr, "Memoria esaurita\n"); exit(EXIT_FAILURE); }
    n->dato = x;
    n->next = *top;                    // sopra la vecchia cima
    *top = n;
}

int pop(Nodo **top)                    // da chiamare solo se la pila non e' vuota
{
    Nodo *vecchio = *top;
    int x = vecchio->dato;
    *top = vecchio->next;              // la cima scende di un nodo
    free(vecchio);
    return x;
}

int isEmpty(const Nodo *top) { return top == NULL; }
```



Pila e coda realizzate con liste concatenate dopo l'inserimento di 1, 2, 3

Code

DESCRIZIONE FORMALE

Una **coda** (queue) è una sequenza in cui gli elementi si inseriscono a un'estremità, la **fine** (tail), e si rimuovono dall'altra, l'**inizio** (head). Segue la politica **FIFO** (First In, First Out): il primo elemento inserito è il primo a uscire, come la fila alla cassa. Le operazioni sono **enqueue** (inserisce in fondo), **dequeue** (rimuove e restituisce il primo elemento), **front** (legge il primo senza rimuoverlo), **isEmpty** e, con un array, **isFull**. Le code servono ovunque ci sia da servire richieste nell'ordine di arrivo: la coda di stampa, i processi in attesa della CPU, i pacchetti in un router, la lettura di dati che vanno elaborati nell'ordine del file.

IMPLEMENTAZIONE CON LISTA

Con una lista si tengono **due puntatori**: alla **testa**, da cui si estrae, e alla **coda** (ultimo nodo), in cui si inserisce. Così entrambe le operazioni costano un passo, senza scorrere la lista. I casi delicati sono quelli in cui la coda è vuota: inserendo in una coda vuota il nuovo nodo diventa sia testa sia coda; estraendo l'ultimo elemento, oltre alla testa bisogna riportare a NULL anche il puntatore alla coda, che altrimenti punterebbe a un nodo liberato.

```
typedef struct {
    Nodo *testa;                      // da qui si estrae
    Nodo *coda;                       // qui si inserisce
} Coda;                              // coda vuota: Coda q = {NULL, NULL};

void enqueue(Coda *q, int x)
{
    Nodo *n = malloc(sizeof(Nodo));
    if (n == NULL) { fprintf(stderr, "Memoria esaurita\n"); exit(EXIT_FAILURE); }
    n->dato = x;
    n->next = NULL;                   // sara' l'ultimo
    if (q->coda == NULL)              // coda vuota: e' anche il primo
        q->testa = n;
    else
        q->coda->next = n;            // aggancia dopo l'ultimo
    q->coda = n;
}

int dequeue(Coda *q)                  // da chiamare solo se la coda non e' vuota
{
    Nodo *primo = q->testa;
    int x = primo->dato;
    q->testa = primo->next;
    if (q->testa == NULL)             // era l'unico elemento
        q->coda = NULL;
    free(primo);
    return x;
}

int isEmptyQ(const Coda *q) { return q->testa == NULL; }
```

IMPLEMENTAZIONE CON ARRAY CIRCOLARE

Con un array non si può semplicemente inserire in fondo e togliere dall'inizio, perché ogni dequeue lascerebbe un buco all'inizio e la coda «scivolerebbe» verso la fine dell'array; far scorrere tutti gli elementi a ogni estrazione sarebbe costoso. Si usa allora un **array circolare**: si tengono l'indice del primo elemento e il numero di elementi, e le posizioni si calcolano **modulo** la dimensione dell'array, in modo che dopo l'ultima cella si ricominci dalla prima. Il prossimo elemento si scrive in posizione `(testa + n) % MAX`, e dopo un'estrazione la testa avanza a `(testa + 1) % MAX`.

```
typedef struct { int dati[MAX]; int testa; int n; } CodaArray;

int enqueueA(CodaArray *q, int x)                    // 0 se la coda e' piena
{
    if (q->n == MAX) return 0;
    q->dati[(q->testa + q->n) % MAX] = x;            // prima cella libera, con giro
    q->n++;
    return 1;
}

int dequeueA(CodaArray *q)                           // coda non vuota
{
    int x = q->dati[q->testa];
    q->testa = (q->testa + 1) % MAX;                 // la testa avanza, con giro
    q->n--;
    return x;
}
```

ESEMPIO SVOLTO

Una coda di stringhe

Si supponga di dover inserire in una coda le parole di un file, una per riga: la coda contiene **stringhe**. Il nodo contiene allora un `char *` allocato con malloc della lunghezza della parola più uno, in cui la parola viene copiata con strcpy. Se ora si vuole estrarre ogni parola e reinserirla in fondo, modificata se ha lunghezza dispari: si estrae e si reinserisce esattamente **n volte**, dove n è il numero di elementi presenti all'inizio, altrimenti le parole reinserite verrebbero elaborate di nuovo. Alla fine la coda contiene le stesse parole nello stesso ordine FIFO. Infine si può svuotare la coda in un array di puntatori a stringa, `char **R = malloc(n * sizeof(char *))`, che viene poi ordinato con il bubble sort usando strcmp.

DA RICORDARE

Che cosa sono pile e code? Come si implementano?

Sono sequenze in cui l'accesso è limitato alle estremità. La pila segue la politica LIFO: inserimento (push) e rimozione (pop) avvengono entrambi in cima, quindi esce per primo l'ultimo elemento inserito; si usa per lo stack delle attivazioni, l'annullamento di operazioni, il controllo delle parentesi. La coda segue la politica FIFO: si inserisce in fondo (enqueue) e si estrae dall'inizio (dequeue), quindi esce per primo il primo inserito; si usa per servire richieste nell'ordine di arrivo. Entrambe si implementano con un array o con una lista concatenata. La pila con array usa un indice top; con lista la cima è la testa. La coda con array usa un array circolare con indici calcolati modulo la dimensione; con lista si tengono un puntatore alla testa per estrarre e uno all'ultimo nodo per inserire. Con l'array la capacità è fissa e va controllato il riempimento; con la lista la dimensione è limitata solo dalla memoria, ma ogni operazione richiede malloc o free.

CAPITOLO 23

Algoritmi di ordinamento

Il problema dell'ordinamento e tre algoritmi classici, Selection Sort, Insertion Sort e Bubble Sort, applicati ad array e liste.

Il problema dell'ordinamento

Ordinare una sequenza di n elementi a_0, a_1, …, a_{n-1} significa trovarne una **permutazione** (cioè una disposizione degli stessi elementi) in cui ogni elemento è minore o uguale al successivo: $a_0 \le a_1 \le \ldots \le a_{n-1}$ per l'ordine crescente. Gli elementi devono essere confrontabili: numeri con `<`, stringhe con `strcmp`, record in base a un campo detto **chiave** (l'età di un artigiano, la media dei pezzi). Ordinare i dati è utile in sé (stampare un elenco in ordine) e rende più efficienti altre operazioni, prima fra tutte la ricerca.

Gli algoritmi di questo capitolo sono **in loco**: ordinano l'array spostando gli elementi al suo interno, senza usare un secondo array. Si confrontano in base al **numero di confronti** e di **spostamenti** che eseguono in funzione di n, e in base alla **stabilità**: un algoritmo è **stabile** se elementi con chiave uguale restano nell'ordine relativo in cui erano, proprietà utile quando si ordinano record. Tutti e tre gli algoritmi richiesti eseguono, nel caso peggiore, un numero di confronti proporzionale a $\mathbf{n^2}$ (si dice che sono **quadratici**, $O(n^2)$): raddoppiando gli elementi il tempo quadruplica. Vanno bene per array piccoli, come quelli dei programmi d'esempio.

Tutti gli algoritmi usano lo **scambio** di due elementi, che conviene scrivere come funzione con il passaggio per riferimento del capitolo 10: `void scambia(int *a, int *b) { int t = *a; *a = *b; *b = t; }`, chiamata con `scambia(&v[i], &v[j])`. Le funzioni di ordinamento ricevono l'array e la sua dimensione e lo modificano direttamente, perché gli array sono passati per riferimento.

Selection Sort

Idea. Si divide l'array in una parte **già ordinata**, a sinistra, inizialmente vuota, e una parte **da ordinare**, a destra. A ogni passo si **seleziona il minimo** della parte da ordinare e lo si **scambia** con il primo elemento di quella parte, che così entra nella parte ordinata nella sua posizione definitiva. Dopo n − 1 passi l'ultimo elemento è necessariamente il massimo e l'array è ordinato. È il modo in cui si ordinerebbe a mano una mano di carte cercando ogni volta la più bassa.

```
void selectionSort(int v[], int n)
{
    for (int i = 0; i < n - 1; i++) {        // v[0..i-1] e' gia' ordinato
        int min = i;                          // indice del minimo di v[i..n-1]
        for (int j = i + 1; j < n; j++)
            if (v[j] < v[min])
                min = j;
        if (min != i)
            scambia(&v[i], &v[min]);          // il minimo va in posizione i
    }
}
```

ESEMPIO SVOLTO

Selection sort su {5, 2, 9, 1, 6}

Passo 1: il minimo di tutto l'array è 1 (posizione 3), scambiato con v[0]: **1** | 2 9 5 6. Passo 2: il minimo di {2, 9, 5, 6} è 2, già in posizione: 1 **2** | 9 5 6. Passo 3: il minimo di {9, 5, 6} è 5, scambiato con 9: 1 2 **5** | 9 6. Passo 4: il minimo di {9, 6} è 6, scambiato con 9: 1 2 5 **6** | 9. Fine: 1 2 5 6 9.

Costo. Il passo i esegue n − 1 − i confronti, qualunque sia l'input: in totale (n − 1) + (n − 2) + … + 1 = **n(n − 1)/2 confronti**, sempre, anche su un array già ordinato. Gli scambi invece sono **al più n − 1**, uno per passo: il selection sort è conveniente quando spostare gli elementi costa molto. Non è stabile, perché lo scambio può portare un elemento oltre un altro uguale. Si può scrivere anche in forma ricorsiva: si mette il minimo in prima posizione e si ordina ricorsivamente il resto dell'array.

Insertion Sort

Idea. È il metodo con cui si ordinano le carte in mano mentre si pescano. La parte sinistra dell'array è ordinata (all'inizio contiene solo il primo elemento). A ogni passo si prende il primo elemento della parte non ordinata, la **chiave**, e lo si **inserisce al posto giusto** nella parte ordinata: si fanno **scorrere di una posizione a destra** tutti gli elementi ordinati maggiori della chiave, e la chiave va nel posto rimasto libero. Dopo n − 1 inserimenti l'array è ordinato.

```
void insertionSort(int v[], int n)
{
    for (int i = 1; i < n; i++) {            // v[0..i-1] e' ordinato
        int chiave = v[i];                    // elemento da inserire
        int j = i - 1;
        while (j >= 0 && v[j] > chiave) {     // sposta a destra i maggiori della chiave
            v[j + 1] = v[j];
            j--;
        }
        v[j + 1] = chiave;                    // la chiave nel posto liberato
    }
}
```

ESEMPIO SVOLTO

Insertion sort su {5, 2, 9, 1, 6}

Passo 1: chiave 2; il 5 scorre a destra e il 2 va in testa: **2 5** | 9 1 6. Passo 2: chiave 9; nessun elemento è maggiore, resta dov'è: 2 5 9 | 1 6. Passo 3: chiave 1; scorrono 9, 5 e 2, e l'1 va in testa: 1 2 5 9 | 6. Passo 4: chiave 6; scorre solo il 9: 1 2 5 6 9. Fine.

Costo. Dipende dall'input. Nel **caso migliore**, un array già ordinato, il ciclo interno non scorre mai e servono solo **n − 1 confronti**. Nel **caso peggiore**, un array ordinato al contrario, ogni chiave deve scorrere fino in testa e i confronti sono n(n − 1)/2, come per il selection sort. È quindi molto efficiente su array **quasi ordinati**. È **stabile**, perché la chiave si ferma appena trova un elemento minore o uguale e non scavalca gli uguali. È lo stesso principio dell'**inserimento in una lista ordinata** della Parte VI.

Bubble Sort

Idea. Si scorre l'array confrontando **coppie di elementi adiacenti** e scambiandoli se sono nell'ordine sbagliato. Al termine della prima passata il **massimo** è arrivato in fondo, «salendo» come una bolla d'aria; dopo la seconda passata il secondo massimo è in penultima posizione, e così via. A ogni passata la parte finale ordinata cresce di un elemento, quindi la passata p può fermarsi a n − p. Bastano al più n − 1 passate. **Ottimizzazione**: se in una passata non avviene nessuno scambio l'array è già ordinato e ci si può fermare, usando una variabile flag.

```
void bubbleSort(int v[], int n)
{
    int scambiato = 1;                          // flag
    for (int passo = 1; passo < n && scambiato; passo++) {
        scambiato = 0;
        for (int i = 0; i < n - passo; i++)     // la parte finale e' gia' ordinata
            if (v[i] > v[i + 1]) {               // coppia in disordine
                scambia(&v[i], &v[i + 1]);
                scambiato = 1;
            }
    }                                           // nessuno scambio: ordinato
}
```

ESEMPIO SVOLTO

Bubble sort su {5, 2, 9, 1, 6}

Passata 1: (5, 2) scambio → 2 5 9 1 6; (5, 9) no; (9, 1) scambio → 2 5 1 9 6; (9, 6) scambio → 2 5 1 6 **9**. Il 9 è in fondo. Passata 2 (fino alla posizione 3): (2, 5) no; (5, 1) scambio → 2 1 5 6 9; (5, 6) no → 2 1 5 **6 9**. Passata 3: (2, 1) scambio → 1 2 5 6 9; (2, 5) no. Passata 4: (1, 2) nessuno scambio, quindi il flag resta a 0 e l'algoritmo si ferma. Risultato: 1 2 5 6 9.

Costo. Nel caso peggiore (array ordinato al contrario) esegue n(n − 1)/2 confronti e altrettanti scambi; con il flag, nel caso migliore (array già ordinato) basta una passata di n − 1 confronti. È **stabile**, perché scambia solo coppie strettamente in disordine. Tra i tre è in genere il più lento, per il gran numero di scambi, ma è il più semplice da scrivere e da adattare.

ORDINARE STRINGHE E RECORD

Lo stesso algoritmo ordina qualunque tipo di dato: basta cambiare il **confronto** e lo **scambio**. Per le stringhe il confronto si fa con `strcmp(s[i], s[i + 1]) > 0`; se l'array contiene **puntatori** alle stringhe (`char *s[]`), lo scambio scambia i puntatori e non i caratteri. Per i record si confronta il campo chiave e si scambiano intere struct, che si possono assegnare. Con un **puntatore a funzione** come parametro (capitolo 17) la stessa funzione di ordinamento può ricevere il criterio di confronto, crescente o decrescente.

```
void bubbleStringhe(char *s[], int n)       // s: array di puntatori a stringhe
{
    for (int passo = 1; passo < n; passo++)
        for (int i = 0; i < n - passo; i++)
            if (strcmp(s[i], s[i + 1]) > 0) {  // ordine lessicografico
                char *t = s[i];                 // scambia i puntatori, non i caratteri
                s[i] = s[i + 1];
                s[i + 1] = t;
            }
}
/* {"mela","banana","kiwi","ranciaa","pera"} -> banana kiwi mela pera ranciaa */
```

Ordinare una lista concatenata

Gli stessi algoritmi si applicano alle liste, con una differenza: non c'è accesso diretto per indice, quindi si lavora con puntatori che **scorrono** i nodi. Il **bubble sort** su lista confronta ogni nodo con il successivo e, se sono in disordine, ne **scambia i dati** (i nodi restano dove sono). Al posto dell'indice n − passo si usa un puntatore `ultimo` al primo nodo della parte già ordinata: a ogni passata si scorre fino a lì, e al termine ultimo si sposta all'indietro sull'ultimo nodo esaminato.

```
void bubbleLista(Nodo *testa)
{
    if (testa == NULL) return;
    Nodo *ultimo = NULL;                       // inizio della parte gia' ordinata
    int scambiato = 1;
    while (scambiato) {
        scambiato = 0;
        Nodo *p = testa;
        while (p->next != ultimo) {            // confronta p con il successivo
            if (p->dato > p->next->dato) {     // scambia i DATI dei due nodi
                int t = p->dato;
                p->dato = p->next->dato;
                p->next->dato = t;
                scambiato = 1;
            }
            p = p->next;
        }
        ultimo = p;                            // p e' ora al suo posto definitivo
    }
}
```

L'**insertion sort** su lista è naturale: si costruisce una nuova lista ordinata staccando uno alla volta i nodi della lista originale e inserendoli con l'inserimento ordinato del capitolo 21; i nodi vengono riusati, senza allocarne di nuovi. Il **selection sort** su lista consiste nell'estrarre ripetutamente il nodo minimo (come `estraiMax` del capitolo 21) e accodarlo a una nuova lista. Quando i dati sono grandi struct conviene spostare i nodi piuttosto che scambiarne i dati.

```
Nodo *insertionLista(Nodo *testa)           // restituisce la nuova testa
{
    Nodo *ordinata = NULL;
    while (testa != NULL) {
        Nodo *n = testa;                       // stacca il primo nodo
        testa = testa->next;
        Nodo *prec = NULL, *curr = ordinata;   // e lo inserisce in ordine
        while (curr != NULL && curr->dato <= n->dato) {   // <= : stabile
            prec = curr;
            curr = curr->next;
        }
        n->next = curr;
        if (prec == NULL) ordinata = n; else prec->next = n;
    }
    return ordinata;
}
```

	SELECTION SORT	INSERTION SORT	BUBBLE SORT
idea	seleziona il minimo e lo mette in testa alla parte da ordinare	inserisce ogni elemento al posto giusto nella parte ordinata	scambia coppie adiacenti: il massimo sale in fondo
confronti, caso peggiore	$n(n-1)/2$	$n(n-1)/2$	$n(n-1)/2$
confronti, array già ordinato	$n(n-1)/2$	$n-1$	$n-1$ (con il flag)
scambi / spostamenti	al più $n-1$	fino a $n(n-1)/2$	fino a $n(n-1)/2$
stabile	no	sì	sì

DA RICORDARE

Descrivere gli algoritmi di ordinamento Selection, Insertion e Bubble Sort e confrontarli.

Selection sort: a ogni passo cerca il minimo della parte non ordinata e lo scambia con il primo elemento di quella parte; esegue sempre $n(n-1)/2$ confronti ma al più $n-1$ scambi, e non è stabile. Insertion sort: prende un elemento alla volta e lo inserisce nella parte ordinata facendo scorrere a destra gli elementi maggiori; nel caso peggiore fa $n(n-1)/2$ confronti, ma su un array già ordinato solo $n-1$, quindi è ottimo su dati quasi ordinati, ed è stabile. Bubble sort: confronta coppie adiacenti e le scambia se in disordine, così a ogni passata il massimo arriva in fondo; nel caso peggiore fa $n(n-1)/2$ confronti e altrettanti scambi, con un flag si ferma appena una passata non fa scambi, ed è stabile. Tutti e tre sono quadratici, operano in loco e si applicano anche alle liste, scambiando i dati dei nodi o spostando i nodi.

APPROFONDIMENTO

Merge Sort

Il merge sort è un algoritmo ricorsivo del tipo «divide et impera»: divide l'array a metà, ordina ricorsivamente le due metà e poi le **fonde** (merge) in un'unica sequenza ordinata, scorrendole in parallelo e prendendo ogni volta il più piccolo dei due elementi in testa. Esegue circa $n \cdot \log_2 n$ confronti, molto meno di n^2 per n grandi (per un milione di elementi circa 20 milioni contro mille miliardi), ma richiede un array di appoggio per la fusione. Non rientra tra gli algoritmi trattati in dettaglio in questa dispensa, ma è il punto di partenza naturale per chi vuole approfondire.

CAPITOLO 24

Algoritmi di ricerca

La ricerca di un elemento in una sequenza: la ricerca sequenziale e la ricerca dicotomica su array ordinati.

Ricerca sequenziale

La **ricerca** consiste nel determinare se un valore, la **chiave**, è presente in una sequenza, e in quale posizione. La **ricerca sequenziale** (o lineare) esamina gli elementi uno alla volta, dal primo, finché non trova la chiave o non arriva alla fine. Funziona su **qualunque** sequenza, ordinata o no, e su array e liste allo stesso modo. Restituisce la posizione della prima occorrenza, oppure −1 (o NULL per le liste) se la chiave non c'è.

```
int ricercaSequenziale(const int v[], int n, int chiave)
{
    for (int i = 0; i < n; i++)
        if (v[i] == chiave)
            return i;                 // trovata: si esce subito
    return -1;                        // esaminati tutti senza trovarla
}
```

Costo. Se la chiave è al primo posto basta un confronto; se non c'è, o è all'ultimo posto, servono **n confronti**; in media, se c'è, circa n/2. Su un array **ordinato** si può fermare la ricerca appena si incontra un elemento maggiore della chiave, perché da lì in poi non potrà più comparire; ma resta proporzionale a n. Varianti tipiche sono la ricerca di tutte le occorrenze (si conta o si stampa invece di uscire), la ricerca del valore più vicino a uno dato (si tiene il minimo della distanza `abs(v[i] − x)`) e la ricerca in una lista, con lo schema `while (p != NULL && p->dato != chiave) p = p->next;`.

Ricerca dicotomica

Se l'array è **ordinato** si può fare molto meglio, come quando si cerca una parola nel dizionario: lo si apre a metà, si guarda se la parola cercata viene prima o dopo, e si scarta l'altra metà. La **ricerca dicotomica** (o binaria) mantiene un intervallo [basso, alto] in cui la chiave può trovarsi, inizialmente tutto l'array. A ogni passo confronta la chiave con l'elemento **centrale** `medio = (basso + alto) / 2`: se sono uguali la chiave è trovata; se la chiave è maggiore può trovarsi solo nella metà destra e si pone basso = medio + 1; se è minore solo nella metà sinistra e si pone alto = medio − 1. Se l'intervallo diventa vuoto (basso > alto) la chiave non c'è.

```
int ricercaBinaria(const int v[], int n, int chiave)     // v deve essere ORDINATO
{
    int basso = 0, alto = n - 1;
    while (basso <= alto) {                    // intervallo non vuoto
        int medio = (basso + alto) / 2;
        if (v[medio] == chiave)
            return medio;                      // trovata
        if (v[medio] < chiave)
            basso = medio + 1;                 // cerca nella meta' destra
        else
            alto = medio - 1;                  // cerca nella meta' sinistra
    }
    return -1;                                 // intervallo vuoto: non c'e'
}

int ricercaBinariaRic(const int v[], int basso, int alto, int chiave)   // ricorsiva
{
    if (basso > alto)                          // caso base: intervallo vuoto
        return -1;
    int medio = (basso + alto) / 2;
    if (v[medio] == chiave)                    // caso base: trovata
        return medio;
    if (v[medio] < chiave)
        return ricercaBinariaRic(v, medio + 1, alto, chiave);
    return ricercaBinariaRic(v, basso, medio - 1, chiave);
}
```

ESEMPIO SVOLTO

Ricerca dicotomica su {1, 3, 4, 7, 9, 11, 15, 20}

Chiave 15. basso = 0, alto = 7, medio = 3, v[3] = 7 < 15: basso = 4. medio = 5, v[5] = 11 < 15: basso = 6. medio = 6, v[6] = 15: trovata in posizione 6 con 3 confronti.

Chiave 5. basso = 0, alto = 7, medio = 3, v[3] = 7 > 5: alto = 2. medio = 1, v[1] = 3 < 5: basso = 2. medio = 2, v[2] = 4 < 5: basso = 3. Ora basso = 3 > alto = 2, l'intervallo è vuoto: la chiave non c'è, si restituisce −1.

Costo. A ogni confronto l'intervallo si **dimezza**, quindi dopo k confronti restano circa $n / 2^k$ elementi; la ricerca termina al più dopo circa $\log_2 n$ confronti. Per 1000 elementi bastano 10 confronti, per un milione 20, contro il milione della ricerca sequenziale nel caso peggiore. La condizione indispensabile è che l'array sia **ordinato**: su un array non ordinato la ricerca dicotomica dà risultati sbagliati. Inoltre richiede l'**accesso diretto** all'elemento centrale, quindi **non si applica alle liste concatenate**, nelle quali raggiungere il nodo centrale richiede comunque di scorrere metà della lista. Se le ricerche sono frequenti conviene quindi tenere i dati ordinati, come nell'insieme su array ordinato della Parte VI.

DA RICORDARE

Che differenza c'è tra ricerca sequenziale e ricerca dicotomica?

La ricerca sequenziale scorre gli elementi uno dopo l'altro finché non trova la chiave o arriva alla fine; funziona su qualunque sequenza, ordinata o no, su array e su liste, ma nel caso peggiore richiede n confronti. La ricerca dicotomica richiede un array ordinato: confronta la chiave con l'elemento centrale e, a seconda del risultato, la trova o scarta metà dell'intervallo, ripetendo finché l'intervallo non è vuoto; servono al più circa $\log_2 n$ confronti (20 per un milione di elementi). Non si può usare su dati non ordinati né sulle liste concatenate, perché richiede l'accesso diretto all'elemento centrale. Si può scrivere in forma iterativa, con gli indici basso e alto, o ricorsiva.