dispense dell'insegnamento di

Programmazione Procedurale

Marco Bernardo Edoardo Bontà

Università degli Studi di Urbino Carlo Bo
Dipartimento di Scienze Pure e Applicate
Scuola di Scienze, Tecnologie e Filosofia dell'Informazione
Corso di Laurea in Informatica – Scienza e Tecnologia (classe L-31)

versione 18/12/2024

Queste dispense sono state preparate con LaTeX e sono reperibili sulla piattaforma Moodle `blended.uniurb.it`
Esse costituiscono soltanto un ausilio per il docente, quindi non sono in nessun modo sostitutive dei testi consigliati.

Si richiede di portare le dispense alle esercitazioni in laboratorio, ma non alle lezioni teoriche in aula.
Se non si capisce un argomento, fare domande a lezione o sul forum di Moodle oppure usufruire del ricevimento.

In aula e in laboratorio, seguire le attività in silenzio per non disturbare e alzare la mano per chiedere la parola.
Quando si arriva tardi a lezione o si prevede di andare via in anticipo, sedersi nei posti vicini all'uscita.

Si consiglia di studiare durante tutto il periodo didattico, evitando di ridursi agli ultimi giorni prima dell'esame.
In ogni caso, è opportuno approfondire sui testi consigliati prima di svolgere il progetto d'esame, non dopo.
Inoltre, conviene avvalersi dei test di autovalutazione della preparazione disponibili sulla piattaforma Moodle.

Indice

Capitolo 1

Introduzione alla Programmazione degli Elaboratori

1.1 Definizioni di Base in Informatica

- Informatica: disciplina che studia l'elaborazione automatica delle informazioni, comprendendo aspetti *scientifici* (Computer Science – teoria della calcolabilità, teoria della complessità, algoritmica, teoria degli automi, linguaggi formali), aspetti *metodologici* (Software Architecture & Engineering) e aspetti *tecnologici* (Information & Communication Technology).
- L'informatica è la scienza che, dato un problema di natura *computazionale*, studia l'esistenza di una soluzione *algoritmica* e, in caso affermativo, determina la quantità di risorse computazionali necessarie in termini di tempo d'esecuzione, spazio di memoria e banda trasmissiva.
- Computer o elaboratore elettronico: insieme di dispositivi elettromeccanici *programmabili* per l'immissione, la memorizzazione, l'elaborazione e l'emissione di informazioni sotto forma di numeri, testi, immagini, audio e video.
- Esistono molti insiemi di dispositivi elettromeccanici che possono svolgere più funzioni (ascensori, elettrodomestici, veicoli, ecc.) ma solo il computer ne può svolgere un numero potenzialmente illimitato, cioè solo il computer è una *macchina programmabile*.
- Hardware: insieme dei dispositivi elettromeccanici che costituiscono un computer, cioè delle risorse computazionali che vengono messe a disposizione (parti fisiche).
- Software: insieme dei programmi eseguibili da un computer, cioè delle istruzioni che vengono impartite alle risorse computazionali per svolgere certi compiti (parti immateriali).
- Esempi di hardware: processore, memoria principale, memoria secondaria (dischi, nastri, ecc.), tastiera, mouse, schermo, stampante, modem, router, access point.
- Esempi di software: sistema operativo, compilatore e debugger per un linguaggio di programmazione, strumento per la scrittura di testi, strumento per fare disegni, visualizzatore di documenti, visualizzatore di immagini, riproduttore di audio e video, foglio elettronico, programma per gestire basi di dati, programma di posta elettronica, navigatore Internet (browser).
- L'informatica ha un fortissimo *impatto socio-economico* dal 1950 derivante dal trasferimento di attività ripetitive o complesse dalle persone alle macchine, dall'eseguibilità di calcoli complicati in tempi brevi e dalla capacità di elaborare, recuperare o trasmettere grandi quantità di informazioni in tempi brevi. Come l'invenzione della stampa nel 1400 ha favorito una maggiore *diffusione della conoscenza* e la rivoluzione industriale nel 1700 ha ampliato le *capacità fisiche* tramite macchine automatiche, così la trasformazione digitale basata sull'elettronica e sull'informatica sta estendendo le *capacità cognitive* attraverso dispositivi programmabili e informazioni propagate in maniera digitale.
- Pensiero computazionale: contributo *culturale* dell'informatica alla società contemporanea, inteso come processo mentale finalizzato alla risoluzione di problemi combinando metodi caratteristici dell'informatica (analisi algoritmica dei problemi, rappresentazione digitale dei dati, automazione delle soluzioni) con capacità intellettuali generali (affrontare la complessità, confrontare le alternative).

1.2 Cenni di Storia dell'Informatica

- Possiamo far risalire la nascita dell'informatica alle prime macchine per automatizzare calcoli numerici. Nel 1642 Pascal costruì una macchina meccanica capace di fare addizioni e sottrazioni. Nel 1672 Leibniz costruì una macchina meccanica capace di fare anche moltiplicazioni e divisioni (precursore delle calcolatrici tascabili a quattro funzioni). Nel 1822 Babbage progettò una macchina meccanica chiamata *Difference Engine* che, utilizzando il metodo delle differenze finite, poteva tabulare funzioni polinomiali e quindi approssimare funzioni logaritmiche e trigonometriche mediante serie di Taylor.
- Nel 1834 Charles Babbage progettò una macchina meccanica *general purpose e programmabile*, chiamata *Analytical Engine*, dotata delle quattro componenti – la sezione di input (lettore di schede perforate), il mulino (unità di computazione), il magazzino (memoria), la sezione di output (output perforato e stampato) – come pure delle istruzioni di base – addizione, sottrazione, moltiplicazione, divisione, trasferimento di dati da/verso memoria e salto (in)condizionato di istruzioni – che troviamo in tutti i moderni computer. Nel 1843 Ada Byron Lovelace tradusse in inglese il saggio di Luigi Menabrea sulla descrizione del progetto di Babbage, arricchendolo con note personali in cui previde con lungimiranza i futuri usi delle macchine programmabili e sviluppò un algoritmo per calcolare i numeri di Bernoulli con l'Analytical Engine, talché viene considerata la prima persona a programmare nella storia.
- Nel 1936 Alan Turing formalizzò il concetto intuitivo di algoritmo mediante un modello operazionale che oggi chiamiamo *macchina di Turing*. Inoltre introdusse l'idea di non costruire più macchine specializzate per scopi specifici, ma di costruire un'unica macchina in grado di svolgere tutti i compiti, che formalizzò attraverso un modello operazionale che oggi chiamiamo *macchina di Turing universale*. I modelli matematici di Turing sono tuttora alla base della teoria della calcolabilità e, a quel tempo, diedero luogo per la prima volta a una visione del software – fino ad allora limitato alle schede perforate introdotte nel 1804 da Jacquard per controllare i telai – inteso come schema di computazione non più cablato nell'hardware, ma indipendente da esso in quanto rappresentato attraverso una delle enumerabili combinazioni di un insieme finito di simboli (i futuri linguaggi di programmazione).
- Nel 1938 Claude Shannon scoprì che il funzionamento dei circuiti elettrici può essere descritto mediante l'algebra sviluppata da Boole verso la metà del 1800 per rappresentare le operazioni della mente umana durante i ragionamenti. Successivamente studiò la codifica e la trasmissione delle informazioni, la cui unità elementare chiamò bit, dando così origine alla teoria dell'informazione e alla moderna crittografia.
- Nel 1943 il governo britannico fece costruire Colossus, il primo elaboratore *elettronico* della storia. Esso aveva l'obiettivo specifico di decifrare i messaggi trasmessi in codice Enigma dalla marina tedesca. Turing partecipò alla sua progettazione contribuendo pertanto alla fine della seconda guerra mondiale.
- Nel 1946 Mauchly ed Eckert costruirono ENIAC, il primo grande elaboratore elettronico *general purpose* della storia, grazie a un finanziamento dell'esercito americano. Pesava 30 tonnellate e occupava una stanza di 9 metri per 15. Era composto da 18000 tubi a vuoto, 70000 resistori, 10000 capacitori e 1500 relè. Aveva 20 registri, ciascuno dei quali poteva contenere un numero decimale a 10 cifre. Veniva programmato impostando migliaia di interruttori e collegando una moltitudine di cavi.
- Nel 1952 John Von Neumann costruì la macchina IAS. Influenzato dal modello della macchina di Turing universale, introdusse l'idea di *computer a programma memorizzato* (dati e programmi caricati insieme in memoria) al fine di evitare la programmazione attraverso l'impostazione di interruttori e cavi. Riconobbe inoltre che la pesante aritmetica decimale seriale usata in ENIAC poteva essere sostituita dalla più efficiente aritmetica binaria parallela. Come nell'Analytical Engine, il cuore della macchina IAS era composto da un'unità di controllo e un'unità aritmetico-logica che interagivano con la memoria e i dispositivi di input/output.
- Nel periodo 1955-1965 vennero costruiti computer basati su transistor, che resero obsoleti i precedenti computer basati su tubi a vuoto (valvole termoioniche). Poco prima dell'esperienza della CEP – Calcolatrice Elettronica Pisana, il primo computer scientifico progettato in Italia grazie all'interessamento di Enrico Fermi e Adriano Olivetti, nel 1957 venne prodotto in Italia dalla Olivetti il primo computer interamente basato su transistor, Elea 9003, progettato dal gruppo di Mario Tchou, seguito all'estero da IBM 7090, DEC PDP-1 e PDP-8, CDC 6600. Nel 1964 venne prodotto in Italia dalla Olivetti il primo personal computer della storia, P101, progettato dal gruppo di Pier Giorgio Perotto.

- Nel periodo 1965-1980 vennero costruiti computer basati su circuiti integrati, cioè circuiti nei quali era possibile includere decine di transistor. Questi computer, tra i quali si annoverano IBM 360 e DEC PDP-11, erano più piccoli, veloci ed economici dei loro predecessori. Nel 1969 nacque Internet. Nel 1971 Federico Faggin progettò presso Intel il primo microprocessore della storia.
- Dal 1980 i computer sono basati sulla tecnologia VLSI, che permette di includere milioni di transistor in un singolo circuito. Iniziò l'era dei personal computer, perché i costi di acquisto diventarono sostenibili anche dai singoli individui (citiamo IBM, Apple, Commodore, Amiga, Atari). Nello stesso tempo le reti di computer – su diverse scale geografiche – diventarono sempre più diffuse. Il continuo aumento del numero di transistor integrabili su un singolo circuito ha determinato il continuo aumento della velocità dei processori e della capacità delle memorie. Tali aumenti sono stati e sono tuttora necessari per far fronte alla crescente complessità delle applicazioni software e dei dati da trattare.
- A partire dal 1950 analoghe evoluzioni hanno avuto luogo nell'ambito dello sviluppo del software, con particolare riferimento a linguaggi e paradigmi di programmazione. Tra questi ultimi, citiamo il *paradigma imperativo di natura procedurale* (Fortran, Cobol, Algol, Basic, Pascal, C, Modula, Ada) o di *natura orientata agli oggetti* (Simula, Smalltalk, C++, Java, C#) e il *paradigma dichiarativo di natura funzionale* (Lisp, Scheme, ML, Haskell) o di *natura logica* (Prolog). ■fltpp_1

1.3 Elementi di Architettura degli Elaboratori

- Per comprendere come i programmi vengono eseguiti, partiamo dall'architettura – ispirata da Babbage, Turing e Von Neumann – di un moderno computer riportata in Fig. 1.1 e dal funzionamento dei suoi componenti fondamentali, cioè processore (o unità centrale di elaborazione) e gerarchia di memorie.
- Computer a programma memorizzato: sia i programmi che i dati devono essere caricati in memoria principale per poter essere elaborati, entrambi rappresentati come sequenze di cifre binarie. Cambiando il programma caricato in memoria principale, cambiano le operazioni effettuate dal computer.
- Passi per l'esecuzione di un programma in un computer a programma memorizzato:
  1. Il programma risiede in memoria secondaria, perché questa è una memoria non volatile.
  2. Il programma viene trasferito dalla memoria secondaria alla memoria principale affinché possa essere eseguito.
  3. Il programma viene eseguito dall'unità centrale di elaborazione. Prima il programma acquisisce i dati di ingresso dai dispositivi periferici di ingresso e/o dalla memoria, poi produce i dati di uscita sui dispositivi periferici di uscita e/o sulla memoria.

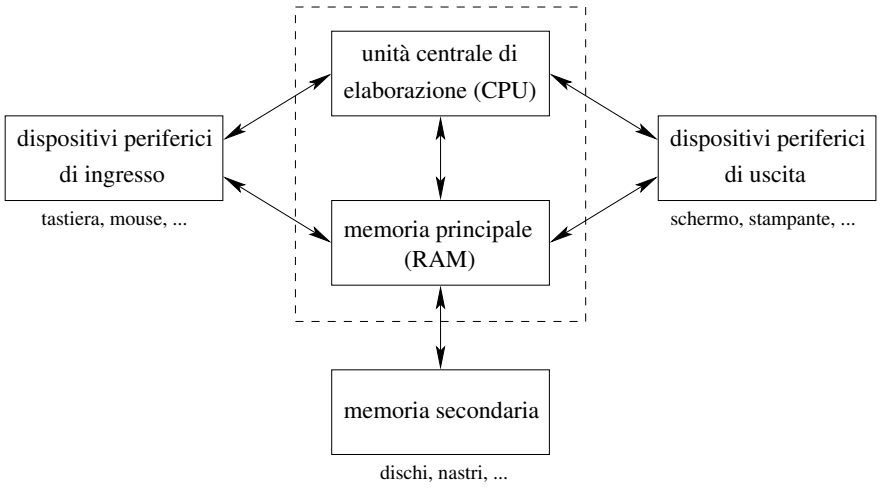


Figura 1.1: Architettura di un computer

- Dal punto di vista di un programma:
  - L'unità centrale di elaborazione è la risorsa che lo esegue.
  - La memoria è la risorsa che lo contiene.
  - I dispositivi periferici di ingresso/uscita sono le risorse che acquisiscono i suoi dati di ingresso e comunicano i suoi dati di uscita.
- L'unità centrale di elaborazione (CPU – central processing unit) svolge i seguenti due compiti:
  - Coordinare tutte le operazioni del computer a livello hardware.
  - Eseguire le operazioni aritmetico-logiche sui dati.
- La CPU è composta dai seguenti dispositivi:
  - Un'unità di controllo (CU – control unit) che, sulla base delle istruzioni di un programma caricato in memoria principale, determina quali operazioni eseguire in quale ordine, trasmettendo gli opportuni segnali di controllo alle altre componenti hardware del computer.
  - Un'unità aritmetico-logica (ALU – arithmetic-logic unit) che esegue le operazioni aritmetico-logiche segnalate dall'unità di controllo sui dati stabiliti dall'istruzione corrente.
  - Un insieme di registri che contengono l'istruzione corrente, i dati correnti e altre informazioni. Questi registri sono piccole unità di memoria che, diversamente dalla memoria principale, hanno una velocità di accesso confrontabile con la velocità dell'unità di controllo.
- Dal punto di vista di un programma caricato in memoria principale, la CPU compie il seguente ciclo di esecuzione dell'istruzione (ciclo fetch-decode-execute):
  1. Trasferire dalla memoria principale all'instruction register (IR) la prossima istruzione da eseguire, la quale è indicata dal registro program counter (PC).
  2. Aggiornare il PC per farlo puntare all'istruzione successiva a quella caricata nell'IR.
  3. Interpretare l'istruzione contenuta nell'IR per determinare quali operazioni eseguire.
  4. Trasferire dalla memoria principale ad appositi registri i dati sui quali opera l'istruzione contenuta nell'IR.
  5. Eseguire le operazioni determinate dall'istruzione contenuta nell'IR (si tratta di una modifica del PC in caso di salto).
  6. Ritornare all'inizio del ciclo.
- Il sistema di numerazione in base 2 è particolarmente adeguato per la rappresentazione delle informazioni nella memoria di un computer. I due valori 0 e 1 sono infatti facilmente associabili a due diverse gamme di valori di tensione come pure a due polarizzazioni opposte in un campo magnetico.
- La memoria è una sequenza di locazioni chiamate celle, ognuna delle quali ha un indirizzo univoco. Il numero di celle è solitamente 2^n, con indirizzi compresi tra 0 e $2^n - 1$ che possono essere rappresentati tramite n cifre binarie.
- Il contenuto di una cella di memoria è una sequenza di cifre binarie chiamate bit (bit = binary digit). Il numero di bit contenuti in una cella è detto dimensione della cella. Una cella di dimensione d può contenere uno fra 2^d valori diversi, compresi fra 0 e $2^d - 1$.
- L'unità di misura della capacità di memoria è il byte (byte = binary octette), dove 1 byte = 8 bit. I suoi multipli più comunemente usati sono kilo-byte (KB), mega-byte (MB), giga-byte (GB) e tera-byte (TB), dove 1 KB = 2^{10} byte, 1 MB = 2^{20} byte, 1 GB = 2^{30} byte e 1 TB = 2^{40} byte.

- La memoria principale (detta anche memoria centrale o primaria) è un dispositivo di memoria con le seguenti caratteristiche:
  - Accesso casuale (RAM – random access memory): il tempo di accesso a una cella non dipende dalla sua posizione fisica.
  - Volatile: il contenuto viene perso quando cessa l'erogazione di energia elettrica.
- La memoria secondaria è un dispositivo di memoria con le seguenti caratteristiche:
  - Accesso sequenziale: il tempo di accesso a una cella dipende dalla sua posizione fisica.
  - Permanente: il contenuto viene mantenuto anche quando cessa l'erogazione di energia elettrica.

  Tipici supporti secondari sono dischi magnetici (rigidi e floppy), nastri magnetici, compact disk e chiavette USB.
- La gerarchia di memorie "registri-memoria principale-memoria secondaria" è caratterizzata da:
  - Velocità decrescenti.
  - Capacità crescenti.
  - Costi decrescenti.
- Dal punto di vista dei programmi:
  - La memoria secondaria li contiene tutti.
  - La memoria principale contiene il programma da eseguire ora.
  - I registri contengono l'istruzione di programma da eseguire ora.

1.4 Elementi di Sistemi Operativi

- Il primo strato di software che viene installato su un computer è il sistema operativo. Esso è un insieme di programmi che gestiscono l'interazione degli utenti del computer con le risorse hardware del computer stesso. Anche la conoscenza della struttura del sistema operativo e del funzionamento dei suoi componenti software è necessaria per comprendere come i programmi vengono eseguiti.
- Negli anni 1960 esistevano pochissimi centri di calcolo, ognuno dotato di un computer molto costoso e di un operatore a cui gli utenti lasciavano i propri programmi e dati per l'esecuzione (*modalità batch*). Il ruolo dei sistemi operativi divenne fondamentale con l'avvento dei sistemi di elaborazione:
  - *multiprogrammati*, in cui più programmi (anziché uno solo) possono risiedere contemporaneamente in memoria principale pronti per l'esecuzione, così da incrementare l'utilizzo del processore in caso di frequenti richieste di input/output da parte del programma in esecuzione grazie alla possibilità di attribuire il processore stesso a qualche altro programma caricato in memoria principale;
  - *time sharing*, in cui più utenti condividono le risorse hardware del centro di calcolo interagendo con il sistema di elaborazione tramite apposite postazioni di lavoro, così da stabilire un dialogo più diretto e rapido tra utente e macchina mediante la turnazione delle risorse (*modalità interattiva*).
- In un ambiente multiprogrammato o time sharing, è necessaria la presenza di un sistema operativo per coordinare le risorse del computer al fine di eseguire correttamente i comandi impartiti dagli utenti direttamente o da programma.
- Obiettivi di un sistema operativo:
  - Dal punto di vista prestazionale, il sistema operativo deve garantire tempi di esecuzione accettabili per i vari programmi caricati in memoria principale che devono essere eseguiti, nonché percentuali di utilizzo adeguate per le varie risorse del computer.
  - Dal punto di vista dell'usabilità, il sistema operativo deve creare un ambiente di lavoro amichevole per gli utenti, al fine di incrementare la loro produttività e il loro grado di soddisfazione.

- Compiti specifici di un sistema operativo:
  - Schedulare i programmi: in che ordine eseguire i programmi pronti per l'esecuzione?
  - Gestire la memoria principale: in che modo caricare i programmi pronti per l'esecuzione?
  - Gestire la memoria secondaria: in che ordine evadere le richieste di accesso?
  - Gestire i dispositivi periferici: in che ordine evadere le richieste di servizio?
  - Trattare i file: come organizzarli sui supporti di memoria secondaria?
  - Preservare l'integrità (errori interni) e la sicurezza (attacchi esterni) dei dati.
  - Acquisire comandi dagli utenti ed eseguirli:
    * Interfaccia a linea di comando (CLI): Unix, Microsoft DOS, Linux.
    * Interfaccia grafica (GUI) dotata di finestre, icone, menù e puntatore (approccio WIMP): Mac OS, Microsoft Windows, Unix con ambiente XWindow, Linux con ambienti KDE o Gnome.

1.5 Elementi di Linguaggi di Programmazione e Compilatori

- La programmazione di un computer richiede un linguaggio in cui esprimere i programmi.
- Il processore di ogni computer è dotato di due linguaggi di programmazione:
  - *Linguaggio macchina*: ogni istruzione è costituita dal codice operativo espresso in binario (addizione, sottrazione, moltiplicazione, divisione, trasferimento dati o salto istruzioni) e dai valori o dagli indirizzi degli operandi espressi in binario. È direttamente interpretabile dal processore.
  - *Linguaggio assemblativo*: ogni istruzione è costituita dal codice operativo espresso in formato mnemonico e dai valori o dagli indirizzi degli operandi espressi in formato mnemonico. Mantiene dunque una corrispondenza uno-a-uno con il linguaggio macchina, ma necessita di un traduttore detto *assemblatore* per rendere i suoi programmi eseguibili dal processore.
- Inconvenienti dei linguaggi macchina e assemblativo:
  - Basso livello di astrazione: è difficile scrivere programmi in questi linguaggi.
  - Dipendenza dall'architettura di un particolare computer: i programmi espressi in questi linguaggi non sono portabili su computer diversi, cioè non possono essere eseguiti su altri computer aventi un processore o un'architettura diversa.
- Caratteristiche dei linguaggi di programmazione di alto livello (sviluppati a partire dal 1955 con Fortran, per le applicazioni scientifiche, e Cobol, per le applicazioni gestionali):
  - Alto livello di astrazione: è più facile scrivere programmi in questi linguaggi perché sono più vicini al linguaggio naturale delle persone e alla notazione matematica di uso comune.
  - Indipendenza dall'architettura di un particolare computer: i programmi espressi in questi linguaggi sono portabili su computer diversi.
  - Necessità di un traduttore nel linguaggio macchina o assemblativo dei vari computer: una singola istruzione di alto livello corrisponde a una sequenza di più istruzioni di basso livello.
- Il traduttore è chiamato *compilatore* o *interprete* a seconda che la traduzione venga effettuata separatamente dall'esecuzione del programma da tradurre (tipico dei linguaggi imperativi, più efficiente) o contestualmente a quest'ultima (tipico dei linguaggi dichiarativi, più interattivo).
- Il compilatore di un linguaggio di programmazione di alto livello per un particolare computer traduce i programmi (comprensibili dagli umani) espressi nel linguaggio di alto livello in programmi equivalenti (eseguibili dal computer) espressi nel linguaggio macchina o assemblativo del computer.

- Componenti di un compilatore e relative fasi di analisi del programma e sintesi del codice:
  - Analizzatore lessicale: organizza la sequenza di simboli presenti nel programma di alto livello in lessemi (come ad esempio parole riservate, identificatori, numeri e simboli specifici) e segnala le sottosequenze di caratteri che non formano lessemi legali del linguaggio.
  - Analizzatore sintattico (o parser): organizza la sequenza precedentemente riconosciuta di lessemi in frasi sulla base delle regole grammaticali del linguaggio (relative per esempio a istruzioni, espressioni, dichiarazioni e direttive) e segnala le sottosequenze di lessemi che violano tali regole.
  - Analizzatore semantico: mediante un sistema di tipi verifica se le varie parti del programma di alto livello hanno un significato compiuto (ad esempio non ha senso scrivere in un programma la somma tra un numero e un vettore di numeri) e segnala eventuali anomalie.
  - Generatore di codice: se non sono stati riscontrati errori, traduce il programma di alto livello in un programma equivalente espresso in linguaggio macchina o assemblativo.
  - Ottimizzatore di codice: modifica le istruzioni macchina o assemblative del programma precedentemente generato al fine di ridurne il tempo di esecuzione e/o la dimensione senza però alterarne la semantica. ■fltpp_2

1.6 Una Metodologia di Sviluppo Software "in the Small"

- Utilizzeremo una metodologia di sviluppo software "in the small" (cioè per programmi di piccole dimensioni) che è composta dalle seguenti fasi in cui si passa dal problema alla sua soluzione informatica:
  1. **Specifica del problema**. Enunciare il problema in maniera chiara e precisa. Questa fase richiede solitamente diverse interazioni con chi ha posto il problema, al fine di evidenziare tutti gli aspetti rilevanti del problema stesso.
  2. **Analisi del problema**. Individuare i dati di ingresso del problema e i dati di uscita del problema, assieme alle principali relazioni intercorrenti tra di essi da sfruttare ai fini della soluzione del problema, astraendo dagli aspetti algoritmici che verranno successivamente progettati così come dal linguaggio implementativo.
  3. **Progettazione dell'algoritmo**. Nel contesto del problema specificato e della sua analisi, illustrare e motivare le principali scelte di progetto (in riferimento a strutture dati utilizzate per la rappresentazione dei dati di ingresso e di uscita, idea della soluzione algoritmica basata sulle relazioni intercorrenti tra i dati anzidetti, ecc.) e riportare l'elenco dei passi principali con eventuali raffinamenti dell'algoritmo creato per risolvere il problema, astraendo dalle caratteristiche del linguaggio di programmazione che verrà impiegato per l'implementazione.
  4. **Implementazione dell'algoritmo**. Tradurre l'algoritmo nel prescelto linguaggio di programmazione di alto livello (passi di scrittura, compilazione e linking del programma).
  5. **Testing del programma**. Effettuare diversi test significativi di esecuzione completa del programma, riportando fedelmente per ciascun test sia i dati di ingresso introdotti che i corrispondenti risultati ottenuti (passi di caricamento ed esecuzione del programma). Se alcuni test danno risultati diversi da quelli attesi, individuarne le cause (passo di debugging del programma) e ritornare alle fasi precedenti per correggere gli errori di malfunzionamento rilevati.
  6. **Manutenzione del programma**. Modificare il programma dopo la sua distribuzione qualora errori d'esecuzione, prestazioni scadenti, falle di sicurezza o limiti di usabilità vengano riscontrati dagli utenti del programma (*manutenzione correttiva*), come pure nel caso si rendano necessari degli aggiornamenti a seguito di nuove esigenze degli utenti o sviluppi tecnologici o normativi (*manutenzione evolutiva*). Questa fase richiede l'adozione nelle fasi precedenti di un appropriato stile di programmazione e la produzione di un'adeguata documentazione interna (commenti) ed esterna (relazione tecnica e manuale d'uso) per il programma.

- Passi del procedimento di scrittura, compilazione, linking, caricamento ed esecuzione di un programma dopo la sua progettazione:

  a. Il programma (o software o codice) viene scritto in un linguaggio di programmazione di alto livello e poi memorizzato in un file detto *file sorgente*.

  b. Il file sorgente viene compilato. Se non ci sono errori lessicali, sintattici e semantici, si prosegue con il passo successivo, altrimenti si torna al passo precedente. In caso di successo viene prodotto un file detto *file oggetto*.

  c. Il file oggetto viene collegato con altri eventuali file oggetto contenenti parti di codice richiamate nel programma originario ma non ivi definite (sono di solito definite nelle cosiddette librerie). Il risultato è un file detto *file eseguibile*.

  d. Il file eseguibile viene caricato in memoria principale.

  e. Il file eseguibile viene eseguito sulla CPU – la quale ripete il ciclo fetch-decode-execute per ogni istruzione – sotto la supervisione del sistema operativo – il quale coordina l'esecuzione di tutti i file eseguibili caricati in memoria principale.

- Il primo passo è svolto tramite uno strumento per la scrittura di testi. Il secondo e il terzo passo sono svolti dal compilatore. Il quarto e il quinto passo hanno luogo all'atto del lancio in esecuzione del programma. Esiste un ulteriore passo, detto debugging, che si rende necessario qualora si manifestino errori durante l'esecuzione del programma, a seguito dei quali bisogna modificare il file sorgente e ripetere i passi dal secondo al quinto per il file sorgente modificato.

- Diversamente dagli errori linguistici rilevati dal compilatore, gli errori di malfunzionamento, come l'ottenimento di risultati diversi da quelli attesi o il mancato ottenimento dei risultati a causa di terminazione prematura o non terminazione dell'esecuzione del programma, non sono accompagnati da una diagnostica adeguata. Il debugger aiuta a individuare le cause di tali errori nel file sorgente tramite la possibilità di eseguire il programma un'istruzione alla volta e di ispezionare il contenuto delle locazioni di memoria.

- I passi precedentemente indicati devono essere preceduti da un'adeguata attività di progettazione. Questo perché la programmazione non può essere improvvisata – soprattutto quando si ha a che fare con lo sviluppo di applicazioni software complesse in gruppi di lavoro – ma deve essere guidata da una metodologia ben precisa.

- Oltre a rilasciare software non sufficientemente testato, le peggiori cose che chi programma possa fare sono scrivere un programma senza averlo prima progettato, non seguire in modo coerente uno stile di programmazione comprensibile e non produrre alcuna documentazione oppure produrla tutta solo alla fine.

Capitolo 2

Programmazione Procedurale: Il Linguaggio ANSI C

2.1 Cenni di Storia del Linguaggio C

- La nascita e l'evoluzione del linguaggio C sono legate a quelle del sistema operativo Unix:
  - Nel 1969 Thompson e Ritchie implementarono la prima versione di Unix per il DEC PDP-11 presso i Bell Lab della AT&T, scrivendolo nel linguaggio assemblativo di quella macchina.
  - Nel 1972 Ritchie mise a punto il linguaggio di programmazione C, il cui scopo era di consentire agli sviluppatori di Unix di implementare e sperimentare le loro idee più agevolmente.
  - Nel 1973 Thompson e Ritchie riscrissero il nucleo di Unix in C. Da allora tutte le chiamate di sistema di Unix vennero definite come funzioni C. Ciò rese di fatto il C il linguaggio da utilizzare per scrivere programmi applicativi in ambiente Unix.
  - Nel 1974 i Bell Lab cominciarono a distribuire Unix alle università, quindi il C iniziò a diffondersi all'interno di una comunità più ampia.
  - Nel 1976 venne implementata la prima versione portabile di Unix (cioè funzionante anche su macchine diverse dal DEC PDP-11), incrementando di conseguenza la diffusione del C.
  - Nel 1983 l'ANSI – American National Standard Institute nominò un comitato per stabilire una definizione del linguaggio C che fosse non ambigua e indipendente dalla macchina.
  - Nel 1988 il comitato concluse i suoi lavori con la produzione del linguaggio ANSI C.
- Il C è un linguaggio di programmazione:
  - di alto livello di astrazione, quindi i programmi C necessitano di essere compilati prima della loro esecuzione e sono portabili su tutti i computer dotati del relativo compilatore C;
  - general purpose, cioè non legato a qualche ambito applicativo in particolare, anche se esso mantiene una certa vicinanza al basso livello di astrazione (data la sua storia) che lo rende particolarmente adeguato per lo sviluppo di sistemi software di base come sistemi operativi e compilatori;
  - *imperativo* di natura *procedurale*, in quanto i programmi C sono sequenze di istruzioni che:
    - prescrivono come modificare il contenuto delle locazioni di memoria;
    - raggruppate in blocchi detti procedure dotati di parametri impostabili di volta in volta.
- I programmi C si compongono di espressioni matematiche e di istruzioni imperative raggruppate in procedure parametrizzate che manipolano dati di vari tipi.
- Sebbene il C sia nato con Unix, esso è ormai supportato da tutti i sistemi operativi di largo uso.

2.2 Formato di un Programma con una Singola Funzione

- Questo è il formato più semplice che un programma C può avere:

```
◁direttive al preprocessore (inclusione di librerie, costanti simboliche)▷
◁intestazione della funzione main▷
{
   ◁variabili della funzione main▷
   ◁istruzioni della funzione main▷
}
```

- L'intestazione della funzione contiene nome, parametri e tipo del risultato della funzione stessa, mentre ciò che è racchiuso tra parentesi graffe costituisce il corpo della funzione.

- Esempio di programma: conversione di miglia in chilometri.

  1. **Specifica del problema.** Convertire una distanza espressa in miglia nell'equivalente distanza espressa in chilometri.
  2. **Analisi del problema.** Per questo semplicissimo problema, l'unico dato di ingresso è rappresentato dalla distanza in miglia, mentre l'unico dato di uscita è rappresentato dalla distanza equivalente in chilometri. L'unica relazione da sfruttare è 1 mi = 1.609 km.
  3. **Progettazione dell'algoritmo.** Data la semplicità del problema, non ci sono particolari scelte di progetto da compiere. I passi dell'algoritmo sono i seguenti:
     - Acquisire la distanza in miglia.
     - Convertire la distanza in chilometri.
     - Comunicare la distanza in chilometri.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*****************************************************************************/
/* programma per convertire miglia in chilometri (versione interattiva) */
/*****************************************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define KM_PER_MI 1.609   /* fattore di conversione miglia-chilometri */

/************************************/
/* definizione della funzione main */
/************************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double miglia;        /* input: distanza in miglia */
  double chilometri;    /* output: distanza in chilometri */

  /* acquisire la distanza in miglia */
  printf("Digita la distanza in miglia: ");
  scanf("%lf",
        &miglia);
```

```
    /* convertire la distanza in chilometri */
    chilometri = miglia * KM_PER_MI;

    /* comunicare la distanza in chilometri */
    printf("La stessa distanza in chilometri e': %f\n",
           chilometri);
    return(0);
}
```

- Notare l'indentazione (cioè la sequenza di spazi iniziali che determinano un rientro verso destra) rispetto alle parentesi graffe, introdotta per favorire la leggibilità del corpo della funzione.
- Ciò che è racchiuso tra `/*` e `*/` costituisce dei commenti interni al programma (che verranno ignorati dal compilatore) usati per esplicitare il legame tra identificatori di costanti e variabili e dati del problema e loro relazioni, oppure tra passi dell'algoritmo e sequenze di istruzioni che li implementano.

2.3 Inclusione di Libreria

- La direttiva di inclusione di libreria ha la seguente sintassi:
  `#include <◁`*file di intestazione di libreria standard*`▷>`
  oppure:
  `#include "◁`*file di intestazione di libreria di chi programma*`▷"`
- Essa imparte al preprocessore C (cioè la parte iniziale del compilatore) il comando di sostituire nel testo del programma la direttiva stessa con il contenuto del file di intestazione, così da permettere l'utilizzo di identificatori di costanti simboliche, tipi, variabili e funzioni definiti/dichiarati nella libreria.

2.4 Funzione `main`

- Ogni programma C deve contenere almeno la funzione `main`, il cui corpo (racchiuso tra parentesi graffe) si compone di una sezione di dichiarazioni di variabili locali e di una sezione di istruzioni che manipolano tali variabili.
- La prima istruzione che viene eseguita in un programma C è la prima istruzione della funzione `main`. Le rimanenti istruzioni vengono poi eseguite una alla volta nell'ordine in cui sono state scritte.

2.5 Identificatori

- Gli identificatori del linguaggio C sono sequenze di lettere, cifre decimali e sottotratti che non iniziano con una cifra decimale. Le lettere minuscole sono considerate diverse dalle corrispondenti lettere maiuscole (case sensitivity).
- Gli identificatori denotano nomi di costanti simboliche (p.e. `KM_PER_MI`), tipi (p.e. `int`, `void`, `double`), variabili (p.e. `miglia`, `chilometri`), funzioni (p.e. `main`, `printf`, `scanf`) e istruzioni.
- Gli identificatori si dividono in riservati (p.e. `int`, `void`, `double`, `return`), standard (p.e. `printf`, `scanf`) e introdotti da chi programma (p.e. `KM_PER_MI`, `miglia`, `chilometri`).
- I seguenti identificatori riservati sono predefiniti dal linguaggio C e hanno un significato fissato, quindi all'interno dei programmi essi non sono utilizzabili per scopi diversi da quelli stabiliti dal linguaggio:

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

- Gli identificatori standard sono definiti all'interno delle librerie standard del linguaggio C, ma possono essere ridefiniti e usati per altri scopi, anche se ciò è fortemente sconsigliato.
- Gli identificatori introdotti da chi programma sono tutti gli altri identificatori che compaiono all'interno di un programma. Ecco alcune regole prescrittive e stilistiche per scegliere buoni identificatori:
  - Devono essere diversi dagli identificatori riservati.
  - È consigliabile che siano diversi anche dagli identificatori standard.
  - È consigliabile dare nomi significativi che ricordino ciò che gli identificatori rappresentano, usando verbi per le funzioni e sostantivi per tutti gli altri identificatori.
  - È consigliabile usare sottotratti quando un identificatore è composto da più (abbreviazioni di) parole.
  - È consigliabile evitare identificatori troppo lunghi o molto simili, in quanto questi aumentano la probabilità di commettere errori di scrittura del programma non rilevabili dal compilatore.
  - È una convenzione comunemente seguita in C quella di usare lettere tutte maiuscole negli identificatori di costanti simboliche e lettere tutte minuscole in tutti gli altri identificatori. ■fltpp_3

2.6 Tipi di Dati Predefiniti: `int`, `double`, `char`

- Ogni identificatore di costante simbolica, variabile o funzione ha un tipo ad esso associato, il quale stabilisce – come se fosse una struttura algebrica – l'insieme di valori che l'identificatore può assumere e l'insieme di operazioni nelle quali l'identificatore può essere coinvolto. Il linguaggio C mette a disposizione tre tipi predefiniti: `int`, `double`, `char`.
- Il tipo `int` denota l'insieme dei numeri interi rappresentabili con un certo numero di bit (sottoinsieme finito di $\mathbb{Z}$). Essi vengono espressi tramite la notazione consueta composta da segno e valore assoluto.
- Il tipo `double` denota l'insieme dei numeri reali rappresentabili con un certo numero di bit (sottoinsieme finito di $\mathbb{R}$). Vengono espressi in virgola fissa (p.e. `13.72`) o in virgola mobile (p.e. `0.1372e2`). Ogni numero in virgola fissa è considerato di tipo `double` anche se la parte frazionaria è nulla (p.e. `-13.0`).
- Il tipo `char` denota l'insieme dei caratteri, i quali vengono espressi racchiusi tra apici (p.e. `'a'`). Tale insieme comprende le 26 lettere minuscole, le 26 lettere maiuscole, le 10 cifre decimali, i simboli di punteggiatura, le varie parentesi, gli operatori aritmetici e relazionali e i caratteri di spaziatura (spazio, tabulazione, andata a capo).
- Un'estensione del tipo `char` è il tipo stringa. Esso denota l'insieme delle sequenze finite di caratteri, ciascuna espressa racchiusa tra virgolette (p.e. `"ciao"`).
- Esiste inoltre il tipo `void`, che viene usato per rappresentare in una funzione l'assenza di parametri o l'assenza di risultati da restituire (vedi Sez. 5.2).

2.7 Funzioni di Libreria per l'Input/Output Interattivo

- In modalità interattiva, un programma C dialoga con l'utente durante la sua esecuzione acquisendo dati tramite tastiera e comunicando risultati tramite schermo. Il file di intestazione di libreria standard che mette a disposizione le relative funzioni è `stdio.h`.
- La funzione di libreria standard per acquisire dati tramite tastiera ha la seguente sintassi:
  `scanf(`◁*stringa formato*$_s$▷`,`
  ◁*sequenza indirizzi variabili*▷`)`
  dove:
  - *stringa formato*$_s$ è una sequenza non vuota racchiusa tra virgolette dei seguenti segnaposto:
    - `%d` per un valore di tipo `int`;
    - `%lf` per un valore di tipo `double` in virgola fissa;
    - `%le` per un valore di tipo `double` in virgola mobile;
    - `%lg` per un valore di tipo `double` in virgola fissa o mobile;
    - `%c` per un valore di tipo `char` (usare la funzione `getchar` per acquisire solo un carattere);
    - `%s` per un valore di tipo stringa (usare `%`◁*numero*▷`s` per limitare il numero di caratteri).

  - *sequenza indirizzi variabili* è una sequenza non vuota di indirizzi di variabili separati da virgola (tali indirizzi sono solitamente ottenuti applicando l'operatore "`&`" agli identificatori delle variabili, vedi Sez. 5.7).
  - L'ordine, il tipo e il numero dei segnaposto presenti nella *stringa formato*$_s$ devono corrispondere all'ordine, al tipo e al numero delle variabili nella *sequenza indirizzi variabili*.

- La semantica della funzione `scanf`, cioè il suo effetto a tempo d'esecuzione, è di acquisire i valori da tastiera e memorizzarli da sinistra a destra nelle variabili della *sequenza indirizzi variabili*:
  - L'introduzione di un valore della sequenza tramite tastiera avviene premendo i tasti corrispondenti e viene terminata (anche nel caso di un valore di tipo carattere o stringa) premendo la barra spaziatrice, il tabulatore o il tasto invio. L'introduzione dell'ultimo (o unico) valore della sequenza deve necessariamente essere terminata premendo il tasto invio.
  - L'acquisizione di un valore di tipo numerico o stringa avviene saltando eventuali spazi, tabulazioni e andate a capo precedentemente digitati. Ciò accade anche per l'acquisizione di un valore di tipo carattere solo se il corrispondente segnaposto `%c` è preceduto da uno spazio nella *stringa formato*$_s$ (nessun salto è possibile quando si usa la funzione `getchar`).

- La funzione `scanf` restituisce il numero di valori acquisiti correttamente rispetto ai segnaposto specificati nella *stringa formato*$_s$ (mentre la funzione `getchar` restituisce il carattere acquisito), il che è molto utile per la validazione stretta degli input (vedi Sez. 4.4). Se i segnaposto sono n, il risultato restituito da `scanf` è un intero compreso tra 0 ed n (dove 0 significa che nessun valore è stato acquisito, 1 significa che solo il primo valore è stato acquisito, e così via). Ulteriori valori eventualmente introdotti dopo l'ultimo valore acquisito correttamente e prima dell'andata a capo finale sono considerati come non ancora acquisiti.

- La funzione di libreria standard per stampare su schermo ha la seguente sintassi:
  `printf(`◁*stringa formato*$_p$▷`,`
  ◁*sequenza espressioni*▷`)`
  dove:
  - *stringa formato*$_p$ è una sequenza non vuota racchiusa tra virgolette di caratteri tra i quali possono comparire anche i seguenti segnaposto:
    - `%d` per un valore di tipo `int`;
    - `%f` per un valore di tipo `double` in virgola fissa;
    - `%e` per un valore di tipo `double` in virgola mobile;
    - `%g` per un valore di tipo `double` in virgola fissa o mobile;
    - `%c` per un valore di tipo `char` (usare la funzione `putchar` per stampare solo un carattere);
    - `%s` per un valore di tipo stringa;

    assieme ai seguenti caratteri speciali:
    - `\n` per un'andata a capo;
    - `\t` per una tabulazione.
  - *sequenza espressioni* è una sequenza eventualmente vuota di espressioni separate da virgola.
  - L'ordine, il tipo e il numero dei segnaposto presenti nella *stringa formato*$_p$ devono corrispondere all'ordine, al tipo e al numero delle espressioni nella *sequenza espressioni*.

- All'atto dell'esecuzione, la funzione `printf` stampa su schermo *stringa formato*$_p$ dopo aver sostituito da sinistra a destra gli eventuali segnaposto con i valori delle espressioni nella *sequenza espressioni* e dopo aver tradotto gli eventuali caratteri speciali (la funzione `putchar` stampa il carattere specificato).

- Formattazione per la stampa allineata di valori di tipo `int` uno sotto l'altro:
  - `%`◁*numero*▷`d` specifica che il valore deve essere stampato su *numero* colonne, includendo l'eventuale segno negativo.
  - Se il valore ha meno cifre delle colonne, esso viene allineato a destra con spazi a precedere se positivo, allineato a sinistra con spazi a seguire se negativo.
  - Se il valore ha più cifre delle colonne, la formattazione viene ignorata.

- Formattazione per la stampa allineata di valori di tipo `double` in virgola fissa uno sotto l'altro:
  - `%`◁*numero*$_1$▷`.`◁*numero*$_2$▷`f` specifica che il valore deve essere stampato su *numero*$_1$ colonne, includendo il punto decimale e l'eventuale segno negativo, delle quali *numero*$_2$ sono riservate alla parte frazionaria (*numero*$_1$ può essere omesso).
  - Se il valore ha meno cifre nella parte intera, esso viene stampato allineato a destra con spazi a precedere.
  - Se il valore ha più cifre nella parte intera, *numero*$_1$ viene ignorato.
  - Se il valore ha meno cifre nella parte frazionaria, vengono aggiunti degli zeri a seguire.
  - Se il valore ha più cifre nella parte frazionaria, questa viene arrotondata.
- Formattazione per la stampa allineata di valori di tipo stringa uno sotto l'altro:
  - `%`◁*numero*▷`s` con *numero* positivo specifica che il valore deve essere stampato su almeno *numero* colonne, con allineamento a sinistra se la lunghezza del valore è minore di *numero*.
  - `%`◁*numero*▷`s` con *numero* negativo specifica che il valore deve essere stampato su almeno `-`*numero* colonne, con allineamento a destra se la lunghezza del valore è minore di `-`*numero*.
- Il segnaposto `%.*s` richiede due espressioni corrispondenti nella `printf`, rispettivamente di tipo `int` e stringa, dove la prima specifica quanti caratteri della seconda devono essere stampati.

2.8 Funzioni di Libreria per l'Input/Output tramite File

- In modalità batch, un programma C acquisisce dati tramite file preparati dall'utente prima dell'esecuzione e comunica risultati tramite altri file che l'utente consulterà dopo l'esecuzione. Il file di intestazione di libreria standard che mette a disposizione le relative funzioni è `stdio.h`.
- I file possono essere gestiti direttamente all'interno del programma come segue:
  - Dichiarare una variabile di tipo puntatore a FILE per ogni file da gestire:
    `FILE *`◁*variabile file*▷`;`
    dove `FILE` è un tipo di dato standard definito in `stdio.h` e `*` denota il tipo puntatore (vedi Sez. 6.11). Tale variabile conterrà l'indirizzo di un'area di memoria detta buffer del file, in cui verranno temporaneamente memorizzate le informazioni lette dal file o da scrivere sul file.
  - Aprire ogni file per creare una corrispondenza tra la variabile precedentemente dichiarata – che ne rappresenta il nome logico – e il suo nome fisico all'interno del file system del sistema operativo. L'operazione di apertura specifica se il file deve essere letto (nel qual caso il file deve esistere):
    ◁*variabile file*▷ `= fopen("`◁*nome file*▷`",`
    `"r");`
    oppure scritto (nel qual caso il precedente contenuto del file, se esistente, viene perso):
    ◁*variabile file*▷ `= fopen("`◁*nome file*▷`",`
    `"w");`
    Se il tentativo di apertura del file fallisce, il risultato della funzione `fopen` che viene assegnato alla variabile è il valore della costante simbolica standard `NULL` definita in `stdio.h`.
  - La funzione di libreria standard per leggere da un file aperto in lettura ha la seguente sintassi:
    `fscanf(`◁*variabile file*▷`,`
    ◁*stringa formato*$_s$▷`,`
    ◁*sequenza indirizzi variabili*▷`)`
    Se `fscanf` incontra il carattere di terminazione file, il risultato che essa restituisce è il valore della costante simbolica standard `EOF` definita in `stdio.h`
  - La funzione di libreria standard per verificare se è stata raggiunta la fine di un file aperto in lettura ha la seguente sintassi:
    `feof(`◁*variabile file*▷`)`

  - La funzione di libreria standard per scrivere su un file aperto in scrittura ha la seguente sintassi:
    `fprintf(`◁*variabile file*▷`,`
    ◁*stringa formato*$_p$▷`,`
    ◁*sequenza espressioni*▷`)`
  - Chiudere ogni file dopo l'ultima operazione di lettura/scrittura su di esso al fine di rilasciare il relativo buffer e rendere definitive eventuali modifiche del contenuto del file:
    `fclose(`◁*variabile file*▷`)`
  - Il numero di file che possono rimanere aperti contemporaneamente durante l'esecuzione del programma è limitato e coincide con il valore della costante simbolica standard `FOPEN_MAX` definita in `stdio.h`. Se all'atto della terminazione del programma ci sono ancora dei file aperti, questi vengono automaticamente chiusi dal sistema operativo.

- In alternativa, se il programma opera in modalità batch su un solo file di input e un solo file di output, i due file possono essere specificati mediante il meccanismo di ridirezione nel comando con il quale il programma viene lanciato in esecuzione:
  ◁*file eseguibile*▷ < ◁*file input*▷ > ◁*file output*▷
  In questo caso, bisogna usare le stesse funzioni di libreria standard illustrate in Sez. 2.7.

- Se nessuno dei due precedenti meccanismi viene usato, si intende che i dati vengano letti dal file `stdin` associato alla tastiera (come fa `scanf`) e che i risultati vengano scritti sul file `stdout` associato allo schermo (come fa `printf`). Questi due file, assieme al file `stderr` sul quale vengono riportati eventuali messaggi di errore a tempo di esecuzione, sono messi a disposizione da `stdio.h`.

- Esempio di programma: conversione di miglia in chilometri usando file.

```
/*****************************************************************/
/* programma per convertire miglia in chilometri (versione batch) */
/*****************************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define KM_PER_MI 1.609   /* fattore di conversione miglia-chilometri */

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  FILE   *file_miglia;       /* input: file contenente la distanza in miglia */
  double  miglia;            /* input: distanza in miglia */
  FILE   *file_chilometri;   /* output: file contenente la distanza in chilometri */
  double  chilometri;        /* output: distanza in chilometri */
```

```
    /* aprire i file */
    file_miglia = fopen("miglia.txt",
                        "r");
    file_chilometri = fopen("chilometri.txt",
                            "w");

    /* acquisire la distanza in miglia */
    fscanf(file_miglia,
           "%lf",
           &miglia);

    /* convertire la distanza in chilometri */
    chilometri = miglia * KM_PER_MI;

    /* comunicare la distanza in chilometri */
    fprintf(file_chilometri,
            "La stessa distanza in chilometri e': %f\n",
            chilometri);

    /* chiudere i file */
    fclose(file_miglia);
    fclose(file_chilometri);
    return(0);
}
```

- Esempio di lancio in esecuzione del programma di Sez. 2.2 con ridirezione dei file standard per `scanf` (da `stdin` a `miglia.txt`) e `printf` (da `stdout` a `chilometri.txt`):

  `conversione_mi_km` < `miglia.txt` > `chilometri.txt` ■fltpp_4

Capitolo 3

Espressioni

3.1 Definizione di Costante Simbolica

- Una costante è un elemento di base in un'espressione del linguaggio C, che può comparire in forma letterale – nel qual caso è espressa tramite il suo valore (p.e. `-13`, `47.5`, `'A'`, `"ciao"`) – oppure in forma simbolica, cioè attraverso un identificatore ad essa associato – che eredita il tipo del suo valore.

- La direttiva di definizione di costante simbolica:
  `#define` ◁*identificatore della costante*▷ ◁*valore della costante*▷
  imparte al preprocessore C il comando di sostituire nel testo del programma ogni occorrenza dell'identificatore della costante con il valore della costante prima di compilare il programma (quindi nessuno spazio di memoria viene riservato per l'identificatore di costante).

- Questo meccanismo consente di raccogliere all'inizio di un programma tutti i valori costanti usati nel programma, dando loro dei nomi da usare all'interno del programma che dovrebbero richiamare ciò che i valori costanti rappresentano. Ciò incrementa la leggibilità del programma e agevola le eventuali successive modifiche di tali valori. Infatti non serve andare a cercare tutte le loro occorrenze all'interno del programma, in quanto basta cambiare le loro definizioni all'inizio del programma (e poi ricompilare).

3.2 Dichiarazione di Variabile

- Una variabile è un elemento di base in un'espressione del linguaggio C, che funge da contenitore per un valore. Diversamente dal valore di una costante, il valore di una variabile può cambiare, anche più volte, durante l'esecuzione del programma.

- La dichiarazione di variabile:
  ◁*tipo*▷ ◁*identificatore della variabile*▷`;`
  che in caso di contestuale inizializzazione diventa:
  ◁*tipo*▷ ◁*identificatore della variabile*▷ `=` ◁*valore iniziale*▷`;`
  associa un nome simbolico alla porzione di memoria necessaria per contenere un valore di un certo tipo, dove sia la dimensione della porzione di memoria che la gamma di valori sono determinati dal tipo.

- Più variabili dello stesso tipo possono essere raccolte in un'unica dichiarazione separando i loro identificatori con delle virgole:
  ◁*tipo*▷ ◁*identificatore della variabile*$_1$▷`,`
  ◁*identificatore della variabile*$_2$▷`,`
  ⋮
  ◁*identificatore della variabile*$_n$▷`;`
  dove gli identificatori delle variabili compaiono allineati su linee separate per aumentare la leggibilità della dichiarazione e per poterli affiancare dai rispettivi commenti (variabile di input, output o lavoro).

3.3 Operatori Aritmetici

- Gli operatori permettono di combinare espressioni per formare espressioni più complesse. Ogni operatore del linguaggio C è formalizzabile attraverso una funzione matematica che va dal prodotto cartesiano di un certo numero di insiemi a un insieme, dove quel numero è chiamato l'arietà dell'operatore (da cui l'operatore è detto unario, binario, ternario, ecc.). Un operatore è detto prefisso, infisso o postfisso a seconda che compaia prima, in mezzo o dopo i suoi operandi.
- Operatori aritmetici unari (prefissi):
  - `+`◁*espressione*▷: valore dell'espressione (questo operatore non viene mai usato).
  - `-`◁*espressione*▷: valore dell'espressione cambiato di segno.
- Operatori aritmetici binari additivi (infissi):
  - ◁$espressione_1$▷ `+` ◁$espressione_2$▷: somma tra i valori delle due espressioni.
  - ◁$espressione_1$▷ `-` ◁$espressione_2$▷: differenza tra i valori delle due espressioni.
- Operatori aritmetici binari moltiplicativi (infissi):
  - ◁$espressione_1$▷ `*` ◁$espressione_2$▷: prodotto tra i valori delle due espressioni.
  - ◁$espressione_1$▷ `/` ◁$espressione_2$▷: quoziente tra i valori delle due espressioni (se il valore di $espressione_2$ è zero, il risultato è `NaN` – not a number).
  - ◁$espressione_1$▷ `%` ◁$espressione_2$▷: resto della divisione tra i valori delle due espressioni intere (se il valore di $espressione_2$ è zero, il risultato è `NaN` – not a number).
- Per ogni operatore abbiamo indicato la sintassi prima dei due punti e la semantica dopo i due punti. Notare lo spazio prima e dopo ogni operatore binario per incrementare la leggibilità delle espressioni. Nel caso degli operatori binari, se non è specificato diversamente si intende che $espressione_1$ viene valutata prima di $espressione_2$, cioè il flusso di valutazione va da sinistra verso destra.

3.4 Operatori Relazionali

- Operatori relazionali d'ordine (binari infissi):
  - ◁$espressione_1$▷ `<` ◁$espressione_2$▷: vero se il valore di $espressione_1$ è minore del valore di $espressione_2$.
  - ◁$espressione_1$▷ `<=` ◁$espressione_2$▷: vero se il valore di $espressione_1$ è minore del o uguale al valore di $espressione_2$.
  - ◁$espressione_1$▷ `>` ◁$espressione_2$▷: vero se il valore di $espressione_1$ è maggiore del valore di $espressione_2$.
  - ◁$espressione_1$▷ `>=` ◁$espressione_2$▷: vero se il valore di $espressione_1$ è maggiore del o uguale al valore di $espressione_2$.
- Operatori relazionali d'uguaglianza (binari infissi):
  - ◁$espressione_1$▷ `==` ◁$espressione_2$▷: vero se i valori delle due espressioni sono uguali.
  - ◁$espressione_1$▷ `!=` ◁$espressione_2$▷: vero se i valori delle due espressioni sono diversi.
- Se vera, l'espressione complessiva ha valore `1`, altrimenti ha valore `0`.

3.5 Operatori Logici

- Non essendoci un tipo specifico, nel linguaggio C i valori di verità falso e vero vengono rappresentati numericamente come segue: zero è interpretato come falso, ogni altro numero è interpretato come vero.
- Analogamente agli operatori relazionali, se l'intera espressione è vera essa vale `1`, altrimenti vale `0`.
- Operatori logici unari (prefissi):
  - `!`◁*espressione*▷: negazione logica del valore dell'espressione (vera se il valore dell'espressione è uguale a zero, falsa altrimenti).
- Operatori logici binari (infissi):
  - ◁*espressione*$_1$▷ `&&` ◁*espressione*$_2$▷: congiunzione logica dei valori delle due espressioni (vera sse il valore di entrambe le espressioni è diverso da zero).
  - ◁*espressione*$_1$▷ `||` ◁*espressione*$_2$▷: disgiunzione logica dei valori delle due espressioni (falsa sse il valore di entrambe le espressioni è uguale a zero).
- Nel linguaggio C l'applicazione degli operatori logici binari è soggetta alla cortocircuitazione:
  - In ◁*espressione*$_1$▷ `&&` ◁*espressione*$_2$▷, se il valore di *espressione*$_1$ è falso, *espressione*$_2$ non viene valutata in quanto si può già stabilire che il valore dell'intera espressione è falso.
  - In ◁*espressione*$_1$▷ `||` ◁*espressione*$_2$▷, se il valore di *espressione*$_1$ è vero, *espressione*$_2$ non viene valutata in quanto si può già stabilire che il valore dell'intera espressione è vero.

3.6 Operatore Condizionale

- L'operatore condizionale (ternario infisso):
  `(`◁*espressione*$_1$▷`)? `◁*espressione*$_2$▷`: `◁*espressione*$_3$▷
  dà come valore quello di *espressione*$_2$ oppure quello di *espressione*$_3$ a seconda che il valore di *espressione*$_1$ sia vero o falso.
- L'operatore condizionale permette di scrivere i programmi in maniera più concisa rispetto alle istruzioni di selezione (vedi Sez. 4.3), però l'uso eccessivo può compromettere la leggibilità dei programmi stessi.
- Esempi:
  - Determinazione del massimo tra i valori delle variabili `x` e `y`:
    ```
    (x > y)? x: y
    ```
  - Biimplicazione logica applicata alle variabili `x` e `y` (attenzione a valori non nulli diversi tra loro):
    ```
    (x == y || (x != 0 && y != 0))? 1: 0
    ```
  - Gestione di singolare e plurale:
    ```
    printf("Hai vinto %d centesim%c.\n",
           importo,
           (importo == 1)? 'o': 'i');
    ```

3.7 Operatori di Assegnamento

- Operatori di assegnamento (binari infissi):
  - ◁*variabile*▷ `=` ◁*espressione*▷: il valore dell'espressione diventa il nuovo valore della variabile (attraverso la sua memorizzazione nella porzione di memoria riservata alla variabile).
  - ◁*variabile*▷ `+=` ◁*espressione*▷: la somma tra il valore corrente della variabile e il valore dell'espressione diventa il nuovo valore della variabile.
  - ◁*variabile*▷ `-=` ◁*espressione*▷: la differenza tra il valore corrente della variabile e il valore dell'espressione diventa il nuovo valore della variabile.

  - ◁*variabile*▷ `*=` ◁*espressione*▷: il prodotto tra il valore corrente della variabile e il valore dell'espressione diventa il nuovo valore della variabile.
  - ◁*variabile*▷ `/=` ◁*espressione*▷: il quoziente tra il valore corrente della variabile e il valore dell'espressione diventa il nuovo valore della variabile.
  - ◁*variabile*▷ `%=` ◁*espressione*▷: il resto della divisione tra il valore corrente della variabile intera e il valore dell'espressione intera diventa il nuovo valore della variabile intera.
- In generale ◁*variabile*▷ ◁*op*▷`=` ◁*espressione*▷ sta per:
  ◁*variabile*▷ `=` ◁*variabile*▷ ◁*op*▷ `(`◁*espressione*▷`)`
  dove le parentesi attorno all'espressione sono necessarie per applicare gli operatori nell'ordine corretto.
- Il valore di un'espressione di assegnamento è il nuovo valore della variabile, il che permette l'assegnamento del valore di una singola espressione a molteplici variabili:
  ◁$variabile_1$▷ `=` ◁$variabile_2$▷ `= ... =` ◁$variabile_n$▷ `=` ◁*espressione*▷
- Diversamente dagli altri operatori binari, quelli di assegnamento hanno sulla sinistra una variabile invece di una generica espressione e determinano un flusso di valutazione che va da destra verso sinistra.
- Il simbolo "`=`", usato in C per denotare l'operatore di assegnamento (invece del più appropriato "`:=`"), non va confuso con il simbolo "`=`" tradizionalmente usato in matematica per denotare l'operatore relazionale di uguaglianza, che in C è rappresentato dal simbolo "`==`".

3.8 Operatori di Incremento/Decremento

- Operatori di incremento/decremento postfissi (unari):
  - ◁*variabile*▷`++`: il valore della variabile, la quale viene poi incrementata di una unità.
  - ◁*variabile*▷`--`: il valore della variabile, la quale viene poi decrementata di una unità.
- Operatori di incremento/decremento prefissi (unari):
  - `++`◁*variabile*▷: il valore della variabile incrementato di una unità.
  - `--`◁*variabile*▷: il valore della variabile decrementato di una unità.
- Gli operatori di incremento/decremento sono applicabili solo a variabili e comportano sempre la variazione di un'unità del valore delle variabili cui sono applicati, come se fossero abbreviazioni di operatori di assegnamento. Da questo punto di vista, l'operatore di incremento equivale a ◁*variabile*▷ `+= 1`, mentre l'operatore di decremento equivale a ◁*variabile*▷ `-= 1`.
- Il valore dell'intera espressione di incremento/decremento cambia a seconda che l'operatore di incremento/decremento sia postfisso o prefisso (importante all'interno di un'espressione più grande).
- Esempi:
  - Consideriamo l'espressione `x += y++` e assumiamo che, prima della sua valutazione, la variabile `x` valga `15` e la variabile `y` valga `3`. Al termine della valutazione, la variabile `y` vale `4` e la variabile `x` vale `18` perché il valore di `y++` è quello di `y` prima dell'incremento.
  - Consideriamo l'espressione `x += ++y` e assumiamo le stesse condizioni iniziali dell'esempio precedente. Al termine della valutazione, la variabile `y` vale `4` (come nell'esempio precedente) ma la variabile `x` vale `19` (diversamente dall'esempio precedente) perché il valore di `++y` è quello di `y` dopo l'incremento.

3.9 Operatore Virgola

- L'operatore virgola (binario infisso):
  ◁*espressione*$_1$▷`,` ◁*espressione*$_2$▷
  dà come valore quello di *espressione*$_2$.
- L'operatore virgola viene usato come separatore in una sequenza di espressioni che debbono essere valutate una dopo l'altra all'interno della medesima istruzione (vedi Sez. 4.4). ■fltpp_5

3.10 Tipo delle Espressioni

- Le espressioni aritmetico-logiche del linguaggio C sono formate da occorrenze dei precedenti operatori (aritmetici, relazionali, logici, condizionale, di assegnamento, di incremento/decremento, virgola) applicati a costanti letterali o simboliche e variabili (e risultati di invocazioni di funzioni – vedi Sez. 5.5) di tipo `int` o `double` (o `char` – vedi Sez. 6.5 – o enumerato – vedi Sez. 6.6 – o puntatore – vedi Sez. 6.11).
- Esempi:
  - Stabilire se `x` e `y` sono entrambe maggiori di `z`: `x > z && y > z`
  - Stabilire se `x` vale `1` oppure `3`: `x == 1 || x == 3`
  - Stabilire se `z` è compresa tra `x` e `y` (assumendo $x \leq y$): `x <= z && z <= y`
  - Stabilire se `z` non è compresa tra `x` e `y`: `!(x <= z && z <= y)`
  - Stabilire se `z` non è compresa tra `x` e `y` in modo diverso: `z < x || y < z`
  - Stabilire se `n` è pari: `n % 2 == 0`
  - Stabilire se `n` è dispari: `n % 2 == 1`
  - Stabilire se `anno` è bisestile: `(anno % 4 == 0 && anno % 100 != 0) || (anno % 400 == 0)`
- Il tipo di un'espressione aritmetico-logica dipende dagli operatori presenti in essa e dal tipo dei relativi operandi. La regola generale di attribuzione del tipo a un'espressione è la seguente:
  - Se tutti i suoi operandi sono di tipo `int`, allora l'espressione è di tipo `int`.
  - Se almeno uno dei suoi operandi è di tipo `double`, allora l'espressione è di tipo `double`.

  Nel caso dell'operatore "`%`", i suoi due operandi devono essere di tipo `int`.
- Fanno eccezione gli operatori di assegnamento, nel qual caso il tipo dell'espressione complessiva coincide col tipo della variabile sulla sinistra che quindi prevale sul tipo dell'espressione sulla destra:
  - Se la variabile è di tipo `double` e l'espressione è di tipo `int`, il tipo dell'espressione viene automaticamente convertito in `double` e il relativo valore viene modificato aggiungendogli una parte frazionaria nulla (`.0`) prima che venga assegnato alla variabile.
  - Se la variabile è di tipo `int` e l'espressione è di tipo `double`, il tipo dell'espressione viene automaticamente convertito in `int` e il relativo valore viene modificato troncandone la parte frazionaria prima che venga assegnato alla variabile.
- Esempi dell'impatto del tipo di un'espressione sul suo valore:
  - Il valore di `5 / 2` è `2`, mentre il valore di `5.0 / 2` oppure `5 / 2.0` oppure `5.0 / 2.0` è `2.5`.
  - Dato l'assegnamento `x = 3.75`, se `x` è di tipo `int` allora il suo nuovo valore è `3`.

3.11 Precedenza e Associatività degli Operatori

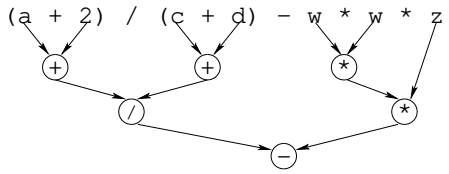
- Al fine di determinare il valore di un'espressione aritmetico-logica, occorre stabilire l'ordine in cui gli operatori (diversi o uguali) presenti nell'espressione debbono essere applicati.
- L'ordine in cui occorrenze di operatori diversi debbono essere applicate è stabilito dalla precedenza degli operatori, di seguito riportata in ordine decrescente:
  - Operatori unari aritmetici, logici, di incremento/decremento: "`+`", "`-`", "`!`", "`++`", "`--`".
  - Operatori aritmetici moltiplicativi: "`*`", "`/`", "`%`".
  - Operatori aritmetici additivi: "`+`", "`-`".
  - Operatori relazionali d'ordine: "`<`", "`>`", "`<=`", "`>=`".
  - Operatori relazionali d'uguaglianza: "`==`", "`!=`".

  - Operatori logici moltiplicativi: “`&&`”.
  - Operatori logici additivi: “`||`”.
  - Operatore condizionale: “`?:`”.
  - Operatori di assegnamento: “`=`”, “`+=`”, “`-=`”, “`*=`”, “`/=`”, “`%=`”.
  - Operatore virgola: “`,`”.

- L’ordine in cui occorrenze dello stesso operatore debbono essere applicate è stabilito dall’associatività dell’operatore; più in generale ciò vale per occorrenze di operatori che hanno la stessa precedenza. Tutti gli operatori riportati sopra sono associativi da sinistra, ad eccezione di quelli di incremento/decremento – che non sono associativi, cioè sono applicabili una sola volta a una variabile – e degli altri unari e di quelli di assegnamento – che sono associativi da destra.

- Le regole di precedenza e associatività possono essere alterate inserendo delle parentesi tonde.

- Un ausilio grafico per determinare l’ordine in cui applicare gli operatori di un’espressione aritmetico-logica è costituito dall’albero di valutazione dell’espressione. Come illustrato nella figura, in questo albero le foglie corrispondono a costanti e variabili (e risultati di invocazioni di funzioni) presenti nell’espressione, mentre i nodi interni corrispondono agli operatori presenti nell’espressione e vengono collocati in base alle regole di precedenza e associatività e alle eventuali parentesi tonde:



- Esempio di programma: determinazione del valore di un insieme di monete.

  1. **Specifica del problema**. Calcolare il valore complessivo di un insieme di monete che un cliente deposita presso una banca, espresso come numero di euro ed eventuale frazione residua di euro.
  2. **Analisi del problema**. I dati di ingresso del problema sono rappresentati dal numero di monete di ogni taglio (1, 2, 5, 10, 20, 50 centesimi e 1, 2 euro) che vengono depositate. I dati di uscita del problema sono rappresentati dal valore complessivo delle monete depositate, espresso in euro e frazione di euro. Le relazioni da sfruttare sono le seguenti: 1 centesimo = 0.01 euro, 2 centesimi = 0.02 euro, 5 centesimi = 0.05 euro, 10 centesimi = 0.10 euro, 20 centesimi = 0.20 euro, 50 centesimi = 0.50 euro, 1 euro = 100 centesimi, 2 euro = 200 centesimi.
  3. **Progettazione dell’algoritmo**. Come scelta di progetto, decidiamo di calcolare prima il valore totale in centesimi perché questo agevola poi la determinazione del valore totale in euro e frazione di euro. I passi sono i seguenti:
     - Acquisire il numero di monete depositate di ogni taglio.
     - Calcolare il valore totale delle monete in centesimi.
     - Convertire il valore totale delle monete in euro e frazione di euro.
     - Comunicare il valore totale delle monete in euro e frazione di euro.
  4. **Implementazione dell’algoritmo**. Questa è la traduzione dei passi in C:

```
/*****************************************************************/
/* programma per determinare il valore di un insieme di monete */
/*****************************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>
```

```
/****************************************/
/* definizione delle costanti simboliche */
/****************************************/

#define VAL_CENT_MONETE_01C   1   /* valore in centesimi di monete da 1 centesimo */
#define VAL_CENT_MONETE_02C   2   /* valore in centesimi di monete da 2 centesimi */
#define VAL_CENT_MONETE_05C   5   /* valore in centesimi di monete da 5 centesimi */
#define VAL_CENT_MONETE_10C  10   /* valore in centesimi di monete da 10 centesimi */
#define VAL_CENT_MONETE_20C  20   /* valore in centesimi di monete da 20 centesimi */
#define VAL_CENT_MONETE_50C  50   /* valore in centesimi di monete da 50 centesimi */
#define VAL_CENT_MONETE_01E 100   /* valore in centesimi di monete da 1 euro */
#define VAL_CENT_MONETE_02E 200   /* valore in centesimi di monete da 2 euro */
#define CENT_PER_EURO       100   /* fattore di conversione centesimi-euro */

/**********************************/
/* definizione della funzione main */
/**********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  int num_monete_01c,     /* input: numero di monete da 1 centesimo */
      num_monete_02c,     /* input: numero di monete da 2 centesimi */
      num_monete_05c,     /* input: numero di monete da 5 centesimi */
      num_monete_10c,     /* input: numero di monete da 10 centesimi */
      num_monete_20c,     /* input: numero di monete da 20 centesimi */
      num_monete_50c,     /* input: numero di monete da 50 centesimi */
      num_monete_01e,     /* input: numero di monete da 1 euro */
      num_monete_02e;     /* input: numero di monete da 2 euro */
  int valore_euro,        /* output: valore espresso in numero di euro */
      frazione_euro;      /* output: numero di centesimi della frazione di euro */
  int valore_centesimi;   /* lavoro: valore espresso in numero di centesimi */

  /* acquisire il numero di monete depositate di ogni taglio */
  printf("Digita il numero di monete da 1 centesimo: ");
  scanf("%d",
        &num_monete_01c);
  printf("Digita il numero di monete da 2 centesimi: ");
  scanf("%d",
        &num_monete_02c);
  printf("Digita il numero di monete da 5 centesimi: ");
  scanf("%d",
        &num_monete_05c);
  printf("Digita il numero di monete da 10 centesimi: ");
  scanf("%d",
        &num_monete_10c);
  printf("Digita il numero di monete da 20 centesimi: ");
  scanf("%d",
        &num_monete_20c);
  printf("Digita il numero di monete da 50 centesimi: ");
  scanf("%d",
        &num_monete_50c);
```

```
  printf("Digita il numero di monete da 1 euro: ");
  scanf("%d",
        &num_monete_01e);
  printf("Digita il numero di monete da 2 euro: ");
  scanf("%d",
        &num_monete_02e);

  /* calcolare il valore totale delle monete in centesimi */
  valore_centesimi = num_monete_01c * VAL_CENT_MONETE_01C +
                     num_monete_02c * VAL_CENT_MONETE_02C +
                     num_monete_05c * VAL_CENT_MONETE_05C +
                     num_monete_10c * VAL_CENT_MONETE_10C +
                     num_monete_20c * VAL_CENT_MONETE_20C +
                     num_monete_50c * VAL_CENT_MONETE_50C +
                     num_monete_01e * VAL_CENT_MONETE_01E +
                     num_monete_02e * VAL_CENT_MONETE_02E;

  /* convertire il valore totale delle monete in euro e frazione di euro */
  valore_euro = valore_centesimi / CENT_PER_EURO;
  frazione_euro = valore_centesimi % CENT_PER_EURO;

  /* comunicare il valore totale delle monete in euro e frazione di euro */
  printf("Il valore delle monete e' di %d euro e %d centesim%c.\n",
         valore_euro,
         frazione_euro,
         (frazione_euro == 1)? 'o': 'i');
  return(0);
}
```

Il programma è ridondante in termini di numero di costanti simboliche definite, numero di variabili di input dichiarate, numero di `printf` e `scanf` utilizzate per acquisire i valori delle variabili di input e numero di addendi presenti nel calcolo del valore totale in centesimi. Per evitare tali ridondanze, bisogna ricorrere a istruzioni di ripetizione (vedi Sez. 4.4) come pure a strutture dati di tipo array (vedi Sez. 6.8) e di tipo stringa (vedi Sez. 6.9). ■fltpp_6

Capitolo 4

Istruzioni

4.1 Istruzione di Assegnamento

- L'istruzione di assegnamento è l'istruzione più semplice del linguaggio C. Sintatticamente, essa è un'espressione di assegnamento terminata da "`;`".
- L'istruzione di assegnamento consente di modificare il contenuto di una porzione di memoria. Questa istruzione è quella fondamentale nel paradigma di programmazione imperativo di natura procedurale. Il motivo è che tale paradigma si basa su modifiche ripetute del contenuto della memoria, il quale è assunto essere lo stato della computazione raggiunto dal programma (vedi Sez. 7.1).

4.2 Istruzione Composta

- Un'istruzione composta del linguaggio C è una sequenza di una o più istruzioni, eventualmente racchiuse tra parentesi graffe. Nel seguito, con *istruzione* intenderemo sempre un'istruzione composta.
- Le istruzioni che formano un'istruzione composta vengono eseguite una alla volta nell'ordine in cui sono state scritte. Per deviare da questo flusso sequenziale, occorre utilizzare istruzioni di controllo del flusso quali le istruzioni di selezione e le istruzioni di ripetizione.

4.3 Istruzioni di Selezione: `if`, `switch`

- Le istruzioni di selezione `if` e `switch` messe a disposizione dal linguaggio C esprimono una scelta tra diverse istruzioni composte, dove la scelta viene operata sulla base del valore di un'espressione aritmetico-logica (scelta deterministica).
- L'istruzione `if` ha diversi formati (nei quali si usa l'indentazione per motivi di leggibilità):
  - Formato con singola alternativa:
    `if (`◁*espressione*▷`)`
    ◁*istruzione*▷
    La sua semantica, cioè il suo effetto a tempo d'esecuzione, è che l'istruzione composta viene eseguita solo se l'espressione è vera.
  - Formato con due alternative:
    `if (`◁*espressione*▷`)`
    ◁*istruzione*$_1$▷
    `else`
    ◁*istruzione*$_2$▷
    Se l'espressione è vera, viene eseguita *istruzione*$_1$, altrimenti viene eseguita *istruzione*$_2$.

  - Formato con più alternative correlate annidate:

    ```
    if (◁espressione1▷)
       ◁istruzione1▷
    else if (◁espressione2▷)
       ◁istruzione2▷
      ⋮
    else if (◁espressione_{n−1}▷)
       ◁istruzione_{n−1}▷
    else
       ◁istruzione_n▷
    ```

    Le espressioni vengono valutate una dopo l'altra nell'ordine in cui sono state scritte, fino a individuarne una che è vera; se questa è *espressione*$_i$, viene eseguita *istruzione*$_i$. Se nessuna delle espressioni è vera, viene eseguita *istruzione*$_n$. Le alternative sono correlate nel senso che tutte le espressioni hanno una parte comune.

- Esempi:
  - Scambio dei valori delle variabili `x` e `y` se il valore della prima è maggiore del valore della seconda:

    ```
    if (x > y)
    {
      tmp = x;
      x = y;
      y = tmp;
    }
    ```

  - Controllo del divisore prima di effettuare una divisione:

    ```
    if (y != 0)
      risultato = x / y;
    else
      printf("Impossibile calcolare il risultato: divisione illegale.\n");
    ```

  - Classificazione del livello di rumore in base agli intervalli interi [0..50], [51..70], [71..90], [91..110]:

    ```
    if (rumore <= 50)
      printf("quiete\n");
    else if (rumore <= 70)            /* non serve verificare rumore >= 51 */
      printf("leggero disturbo\n");
    else if (rumore <= 90)            /* non serve verificare rumore >= 71 */
      printf("disturbo\n");
    else if (rumore <= 110)           /* non serve verificare rumore >= 91 */
      printf("forte disturbo\n");
    else
      printf("rumore insopportabile\n");
    ```

- Nel formato generale, un'istruzione `if` può contenere altre istruzioni `if` arbitrariamente annidate, nel qual caso il numero di occorrenze di `else` non può superare il numero di occorrenze di `if`. La regola di associazione degli `else` agli `if` stabilisce che ogni `else` venga associato all'`if` più vicino che lo precede tra quelli pendenti, cioè quelli non ancora associati e non racchiusi tra parentesi graffe.

- Esempi:
  - Uso errato delle parentesi graffe:

    ```
    if (y == 0)
    {
      printf("E' impossibile calcolare il risultato ");
      printf("perche' e' stata incontrata una divisione ");
    }
    printf("nella quale il divisore e' zero.\n");
    ```

– Associazione dell'`else` all'`if` giusto:

```
if (x == 0)                if (x == 0)                if (x == 0)
  if (y >= 0)              {                          {
    x += y;                  if (y >= 0)                if (y >= 0)
  else                         x += y;                    x += y;
    x -= y;                  else                     }
                               x -= y;                else
                           }                            x -= y;
```

La prima istruzione è equivalente alla seconda, ma non alla terza.

- L'istruzione `switch` ha la seguente sintassi (notare l'allineamento delle clausole `case` e l'indentazione all'interno di ciascun insieme di clausole `case` per motivi di leggibilità):

```
switch (◁espressione▷)
{
  case ◁valore_{1,1}▷:
  case ◁valore_{1,2}▷:
   ...
  case ◁valore_{1,m_1}▷:
    ◁istruzione_1▷
    break;
  case ◁valore_{2,1}▷:
  case ◁valore_{2,2}▷:
   ...
  case ◁valore_{2,m_2}▷:
    ◁istruzione_2▷
    break;
   ⋮
  case ◁valore_{n,1}▷:
  case ◁valore_{n,2}▷:
   ...
  case ◁valore_{n,m_n}▷:
    ◁istruzione_n▷
    break;
  default:
    ◁istruzione_{n+1}▷
    break;
}
```

- L'espressione su cui si basa la selezione deve essere di tipo `int` o `char` (o enumerato – vedi Sez. 6.6). Se il suo valore è uguale a uno dei valori indicati in una delle clausole `case`, tutte le istruzioni composte che seguono quella clausola `case` vengono eseguite fino a incontrare un'istruzione `break` (o la fine dell'istruzione `switch`). Se invece il valore dell'espressione è diverso da tutti i valori indicati nelle clausole `case`, viene eseguita l'istruzione composta specificata nella clausola opzionale `default`.

- La presenza di un'istruzione `break` dopo ogni istruzione composta dell'istruzione `switch` garantisce la corretta strutturazione dell'istruzione `switch` stessa, in quanto permette a ogni istruzione composta di essere eseguita solo se il valore dell'espressione è uguale al valore di una delle clausole `case` associate all'istruzione composta medesima. Ogni istruzione composta ha quindi un unico punto di ingresso – l'insieme delle clausole `case` che immediatamente la precedono – e un unico punto di uscita – l'istruzione `break` che immediatamente la segue (vedi Sez. 4.6).

- La precedente istruzione `switch` è equivalente alla seguente istruzione `if` con alternative correlate:

```
if (◁espressione▷ == ◁valore_{1,1}▷ ||
    ◁espressione▷ == ◁valore_{1,2}▷ ||
     ...
    ◁espressione▷ == ◁valore_{1,m_1}▷)
  ◁istruzione_1▷
else if (◁espressione▷ == ◁valore_{2,1}▷ ||
         ◁espressione▷ == ◁valore_{2,2}▷ ||
          ...
         ◁espressione▷ == ◁valore_{2,m_2}▷)
  ◁istruzione_2▷
 ⋮
else if (◁espressione▷ == ◁valore_{n,1}▷ ||
         ◁espressione▷ == ◁valore_{n,2}▷ ||
          ...
         ◁espressione▷ == ◁valore_{n,m_n}▷)
  ◁istruzione_n▷
else
  ◁istruzione_{n+1}▷
```

- L'istruzione `switch` è quindi più leggibile ma meno espressiva dell'istruzione `if`. Infatti, l'espressione di selezione dell'istruzione `switch` può essere solo di tipo `int` o `char` (o enumerato – vedi Sez. 6.6), può essere confrontata solo attraverso l'operatore relazionale di uguaglianza e gli elementi con cui effettuare i confronti possono essere solo delle costanti (letterali o simboliche).

- Esempio di riconoscimento dei colori della bandiera italiana usando le iniziali maiuscole o minuscole dei colori stessi:

```
char colore;

switch (colore)
{
  case 'V':
  case 'v':
    printf("colore presente nella bandiera italiana: verde\n");
    break;
  case 'B':
  case 'b':
    printf("colore presente nella bandiera italiana: bianco\n");
    break;
  case 'R':
  case 'r':
    printf("colore presente nella bandiera italiana: rosso\n");
    break;
  default:
    printf("colore non presente nella bandiera italiana\n");
    break;
}
```

■fltpp_7

- Esempio di programma: calcolo della bolletta dell'acqua.
  1. **Specifica del problema**. Calcolare la bolletta dell'acqua per un utente sulla base di una quota fissa di 15 euro e una quota variabile di 2.50 euro per ogni metro cubo d'acqua consumato nell'ultimo periodo, più una mora di 10 euro per eventuali bollette non pagate relative a periodi precedenti. Evidenziare l'eventuale applicazione della mora.
  2. **Analisi del problema**. I dati di ingresso del problema sono rappresentati dalle letture del contatore alla fine del periodo precedente e alla fine del periodo corrente (cui la bolletta si riferisce) e dall'importo di eventuali bollette precedenti ancora da pagare. I dati di uscita del problema sono rappresentati dall'importo della bolletta del periodo corrente, evidenziando anche l'eventuale applicazione della mora. Le relazioni da sfruttare sono che il consumo di acqua nel periodo corrente è dato dalla differenza tra le ultime due letture del contatore e che l'importo della bolletta è dato dalla somma della quota fissa, del costo al metro cubo moltiplicato per il consumo di acqua nel periodo corrente, dell'importo di eventuali bollette arretrate e dell'eventuale mora.
  3. **Progettazione dell'algoritmo**. Non ci sono particolari scelte di progetto da compiere. Osservato che l'importo della bolletta è diverso a seconda che vi siano bollette precedenti ancora da pagare o meno, i passi sono i seguenti:
     - Acquisire le ultime due letture del contatore.
     - Acquisire l'importo di eventuali bollette precedenti ancora da pagare.
     - Calcolare l'importo della bolletta:
       - Calcolare l'importo derivante dal consumo di acqua nel periodo corrente.
       - Determinare l'applicabilità della mora.
       - Sommare le varie voci.
     - Comunicare l'importo della bolletta evidenziando l'eventuale mora.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*****************************************************/
/* programma per calcolare la bolletta dell'acqua */
/*****************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define QUOTA_FISSA  15.00   /* quota fissa */
#define COSTO_PER_M3  2.50   /* costo per metro cubo */
#define MORA         10.00   /* mora */

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double lettura_prec,        /* input: lettura alla fine periodo precedente */
         lettura_corr,        /* input: lettura alla fine periodo corrente */
         importo_arretrato;   /* input: importo delle bollette arretrate */
  double importo_bolletta;    /* output: importo della bolletta */
  double importo_consumo,     /* lavoro: importo del consumo */
         importo_mora;        /* lavoro: importo della mora se dovuta */
```

```
    /* acquisire le ultime due letture del contatore */
    printf("Digita il consumo risultante dalla lettura precedente: ");
    scanf("%lf",
          &lettura_prec);
    printf("Digita il consumo risultante dalla lettura corrente: ");
    scanf("%lf",
          &lettura_corr);

    /* acquisire l'importo di eventuali bollette precedenti ancora da pagare */
    printf("Digita l'importo di eventuali bollette precedenti ancora da pagare: ");
    scanf("%lf",
          &importo_arretrato);

    /* calcolare l'importo derivante dal consumo di acqua nel periodo corrente */
    importo_consumo = (lettura_corr - lettura_prec) * COSTO_PER_M3;

    /* determinare l'applicabilita' della mora */
    importo_mora = (importo_arretrato > 0)?
                    MORA:
                    0;

    /* sommare le varie voci */
    importo_bolletta = QUOTA_FISSA +
                       importo_consumo +
                       importo_arretrato +
                       importo_mora;

    /* comunicare l'importo della bolletta evidenziando l'eventuale mora */
    printf("\nTotale bolletta: %.2f euro.\n",
           importo_bolletta);
    if (importo_mora > 0)
    {
      printf("\nLa bolletta comprende una mora di %.2f euro",
             importo_mora);
      printf(" per un arretrato di %.2f euro.\n",
             importo_arretrato);
    }
    return(0);
}
```

4.4 Istruzioni di Ripetizione: `while`, `do-while`, `for`

- Le istruzioni di ripetizione `while`, `do-while` e `for` messe a disposizione dal linguaggio C esprimono l'esecuzione ripetuta di un'istruzione composta, dove la terminazione dell'iterazione dipende dal valore di un'espressione aritmetico-logica detta condizione di continuazione.

- Sintassi dell'istruzione `while` (l'indentazione è per motivi di leggibilità):

  `while (`◁*espressione*▷`)`
  ◁*istruzione*▷

  La sua semantica, cioè il suo effetto a tempo d'esecuzione, è che l'istruzione composta che si trova all'interno viene eseguita finché l'espressione è vera. Se all'inizio l'espressione è falsa, l'istruzione composta non viene eseguita affatto.

- Sintassi dell'istruzione `do-while` e sua equivalente `while` (l'indentazione è per motivi di leggibilità):

```
do                                  ◁istruzione▷
   ◁istruzione▷                     while (◁espressione▷)
while (◁espressione▷);                 ◁istruzione▷
```

  L'istruzione `do-while` è una variante dell'istruzione `while` da usare quando l'istruzione composta al suo interno viene sicuramente eseguita almeno una volta, così da evitare codice ridondante.

- Sintassi dell'istruzione `for` e sua equivalente `while` (l'indentazione è per motivi di leggibilità):

```
for (◁espressione1▷;                ◁espressione1▷;
     ◁espressione2▷;                while (◁espressione2▷)
     ◁espressione3▷)                {
   ◁istruzione▷                       ◁istruzione▷
                                      ◁espressione3▷;
                                      }
```

  L'istruzione `for` è una variante articolata dell'istruzione `while`, dove $espressione_1$ è l'espressione di inizializzazione delle variabili (di controllo del ciclo) presenti nella condizione di continuazione, $espressione_2$ è la condizione di continuazione ed $espressione_3$ è l'espressione di aggiornamento delle variabili (di controllo del ciclo) presenti nella condizione di continuazione.

- Esempio di verifica di primalità di un numero naturale (cioè ≥ 2 e divisibile solo per 1 e per se stesso):

```
for (primo = (n == 2 || (n >= 3 && n % 2 == 1)),   /* nessun pari n > 2 è primo */
       div = 3;                    /* primo possibile divisore dispari di n */
     (primo && div * div <= n);    /* basta esaminare divisori fino alla radice quadrata di n */
     primo = (n % div != 0),       /* n non è primo se risulta divisibile per div */
       div += 2);                  /* prossimo possibile divisore dispari di n */
```

- Quando si usano istruzioni di ripetizione, è fondamentale definire le loro condizioni di continuazione in maniera tale da evitare iterazioni senza termine (ricordiamo che Turing dimostrò che il problema della terminazione non è decidibile), come pure racchiudere le loro istruzioni composte tra parentesi graffe se queste istruzioni sono formate da più di un'istruzione.

- Esistono diversi tipi di controllo della ripetizione:
  - Ripetizione controllata tramite contatore (quando si conosce a priori il numero di iterazioni).
  - Ripetizione controllata tramite sentinella (quando non si conosce il numero di iterazioni).
  - Ripetizione controllata tramite fine file (caso particolare del precedente).
  - Ripetizione relativa all'acquisizione con validazione di uno o più valori.

- Esempi:
  - Uso di un contatore nel calcolo della media di un insieme finito e non vuoto di numeri naturali:

```
for (contatore_valori = 1,
       somma_valori = 0;
     (contatore_valori <= numero_valori);
     contatore_valori++,
       somma_valori += valore)
{
  printf("Digita il prossimo valore: ");
  scanf("%d",
        &valore);
}
printf("La media e': %d.\n",
       somma_valori / numero_valori);
```

– Uso di un valore sentinella nel calcolo della media di un insieme di numeri naturali:

```
numero_valori = somma_valori = 0;
do
{
  printf("Digita il prossimo valore (negativo per terminare): ");
  scanf("%d",
        &valore);
  if (valore >= 0)
  {
    numero_valori++;
    somma_valori += valore;
  }
}
while (valore >= 0);
printf("La media e': %d.\n",
       somma_valori / numero_valori);
```

– Uso di fine file nel calcolo della media di un insieme di numeri naturali memorizzati su file:

```
for (numero_valori = somma_valori = 0;   /* for (numero_valori = somma_valori = 0; */
     (!feof(file_valori));               /*      (fscanf(file_valori,              */
     numero_valori++,                    /*              "%d",                     */
       somma_valori += valore)           /*              &valore) != EOF);         */
  fscanf(file_valori,                    /*      numero_valori++,                  */
         "%d",                           /*        somma_valori += valore);        */
         &valore);                       /*                                        */
printf("La media e': %d.\n",             /* printf("La media e': %d.\n",           */
       somma_valori / numero_valori);    /*        somma_valori / numero_valori);  */
```

– Validazione lasca di un valore acquisito da tastiera:

```
do
{
  printf("Digita il numero di valori di cui calcolare la media (> 0): ");
  scanf("%d",
        &numero_valori);
}
while (numero_valori <= 0);
```

– Validazione stretta dello stesso valore sfruttando il risultato della funzione `scanf` per eliminare dal buffer di `stdin`, tramite `getchar` ripetute fino all'andata a capo finale, eventuali valori non conformi al segnaposto nonché eventuali ulteriori valori (questi valori sono considerati come non acquisiti e quindi determinerebbero la non terminazione del ciclo se non venissero rimossi):

```
int esito_lettura,         /* lavoro: esito della scanf */
    acquisizione_errata;   /* lavoro: esito complessivo dell'acquisizione */

do
{
  printf("Digita il numero di valori di cui calcolare la media (> 0): ");
  esito_lettura = scanf("%d",
                        &numero_valori);
  acquisizione_errata = esito_lettura != 1 || numero_valori <= 0;
  if (acquisizione_errata)
    printf("Valore non accettabile!\n");
  while (getchar() != '\n');
}
while (acquisizione_errata);
```

■fltpp_8

- Esempio di programma: calcolo dei livelli di radiazione.

  1. **Specifica del problema**. In un laboratorio può verificarsi una perdita di un materiale pericoloso, il quale produce un certo livello iniziale di radiazione che poi si dimezza ogni tre giorni. Calcolare il livello delle radiazioni ogni tre giorni, fino a raggiungere il giorno in cui il livello delle radiazioni scende al di sotto di un decimo del livello di sicurezza quantificato in 0.466 mrem.
  2. **Analisi del problema**. I dati di ingresso del problema sono rappresentati dal livello iniziale delle radiazioni. I dati di uscita del problema sono rappresentati dal giorno in cui viene ripristinata una situazione di sicurezza, con il valore del livello delle radiazioni calcolato ogni tre giorni sino a quel giorno. La relazione da sfruttare è il dimezzamento del livello delle radiazioni ogni tre giorni.
  3. **Progettazione dell'algoritmo**. Non ci sono particolari scelte di progetto da compiere. Osservato che il problema richiede il calcolo ripetuto del livello delle radiazioni sino a quando tale livello non scende sotto una certa soglia, i passi sono i seguenti:
     - Acquisire il livello iniziale delle radiazioni.
     - Finché il livello delle radiazioni non scende al di sotto di un decimo del livello di sicurezza, calcolare e comunicare il livello delle radiazioni ogni tre giorni
     - Comunicare il giorno in cui il livello delle radiazioni scende al di sotto di un decimo del livello di sicurezza.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*****************************************************/
/* programma per calcolare i livelli di radiazione */
/*****************************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define SOGLIA_SICUREZZA  0.0466   /* soglia di sicurezza */
#define FATTORE_RIDUZIONE 2        /* fattore di riduzione delle radiazioni */
#define NUMERO_GIORNI     3        /* numero di giorni di riferimento */

/***********************************/
/* definizione della funzione main */
/***********************************/

int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double livello_iniziale;   /* input: livello iniziale delle radiazioni */
  int    giorno_sicurezza;   /* output: giorno di ripristino della sicurezza */
  double livello_corrente;   /* output: livello corrente delle radiazioni */
```

```
  /* acquisire il livello iniziale delle radiazioni */
  do
  {
    printf("Digita il livello iniziale delle radiazioni (> 0): ");
    scanf("%lf",
          &livello_iniziale);
  }
  while (livello_iniziale <= 0);

  /* finche' il livello non scende al di sotto di un decimo del livello di sicurezza,
     calcolare e comunicare il livello delle radiazioni ogni tre giorni */
  for (livello_corrente = livello_iniziale,
         giorno_sicurezza = 0;
       (livello_corrente >= SOGLIA_SICUREZZA);
       livello_corrente /= FATTORE_RIDUZIONE,
         giorno_sicurezza += NUMERO_GIORNI)
    printf("Il livello delle radiazioni al giorno %3d e' %9.4f mrem.\n",
           giorno_sicurezza,
           livello_corrente);

  /* comunicare il giorno in cui il livello delle radiazioni scende al di sotto
     di un decimo del livello di sicurezza */
  printf("Giorno in cui si puo' tornare in laboratorio: %d.\n",
         giorno_sicurezza);
  return(0);
}
```

4.5 Istruzione `goto`

- Il flusso di esecuzione delle istruzioni di un programma C può essere modificato in maniera arbitraria tramite la seguente istruzione:

  `goto` ◁*etichetta*▷`;`

  la quale fa sì che la prossima istruzione da eseguire non sia quella ad essa immediatamente successiva nel testo del programma, ma quella prefissata da:

  ◁*etichetta*▷`:`

  dove *etichetta* è un identificatore.

- Il C eredita l'istruzione `goto` dai linguaggi (macchina e) assemblativi, nei quali non sono disponibili istruzioni di controllo del flusso di esecuzione di alto livello di astrazione – come quelle di selezione e ripetizione, che modificano il flusso di esecuzione in modo controllato – ma solo istruzioni di salto incondizionato e condizionato, che modificano il flusso di esecuzione in modo arbitrario.

- Esempio di uso di istruzioni di salto incondizionato e condizionato per comunicare se il valore di una variabile è pari o dispari:

```
(1)                if (n % 2 == 0)
(2)                  goto scrivi_pari;
(3)                printf("Il numero e' dispari.\n");
(4)                goto continua;
(5)  scrivi_pari:  printf("Il numero e' pari.\n");
(6)  continua:     ...
```

  Se `n` è pari, allora il flusso di esecuzione è `(1)-(2)-(5)-(6)`, altrimenti è `(1)-(3)-(4)-(6)`.

- L'uso dell'istruzione `goto` rende i programmi più difficili da leggere e da mantenere, quindi va evitato (vedremo in Sez. 4.6 che è sempre possibile farlo nei moderni linguaggi di programmazione).

4.6 Teorema Fondamentale della Programmazione Strutturata

- I programmi sono rappresentabili graficamente attraverso schemi di flusso, i quali sono costituiti dai blocchi e dai nodi mostrati in Fig. 4.1.

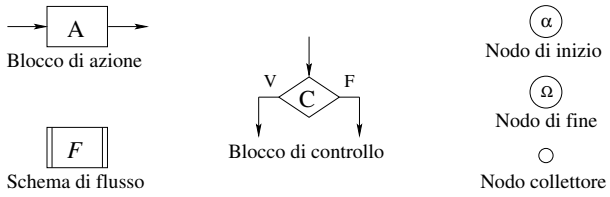


Figura 4.1: Blocchi e nodi degli schemi di flusso

- L'insieme degli schemi di flusso strutturati è definito per induzione sulla struttura grafica degli schemi di flusso come il più piccolo insieme di schemi di flusso tale che:
  - Il primo schema di flusso in Fig. 4.2 è strutturato.
  - Se F_1, F_2 ed F sono schemi di flusso strutturati dai quali sono stati tolti il nodo di inizio e il nodo di fine, allora ciascuno degli altri tre schemi di flusso in Fig. 4.2 è strutturato.

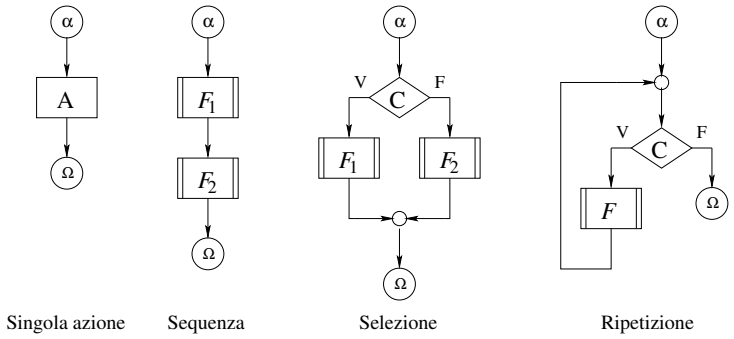
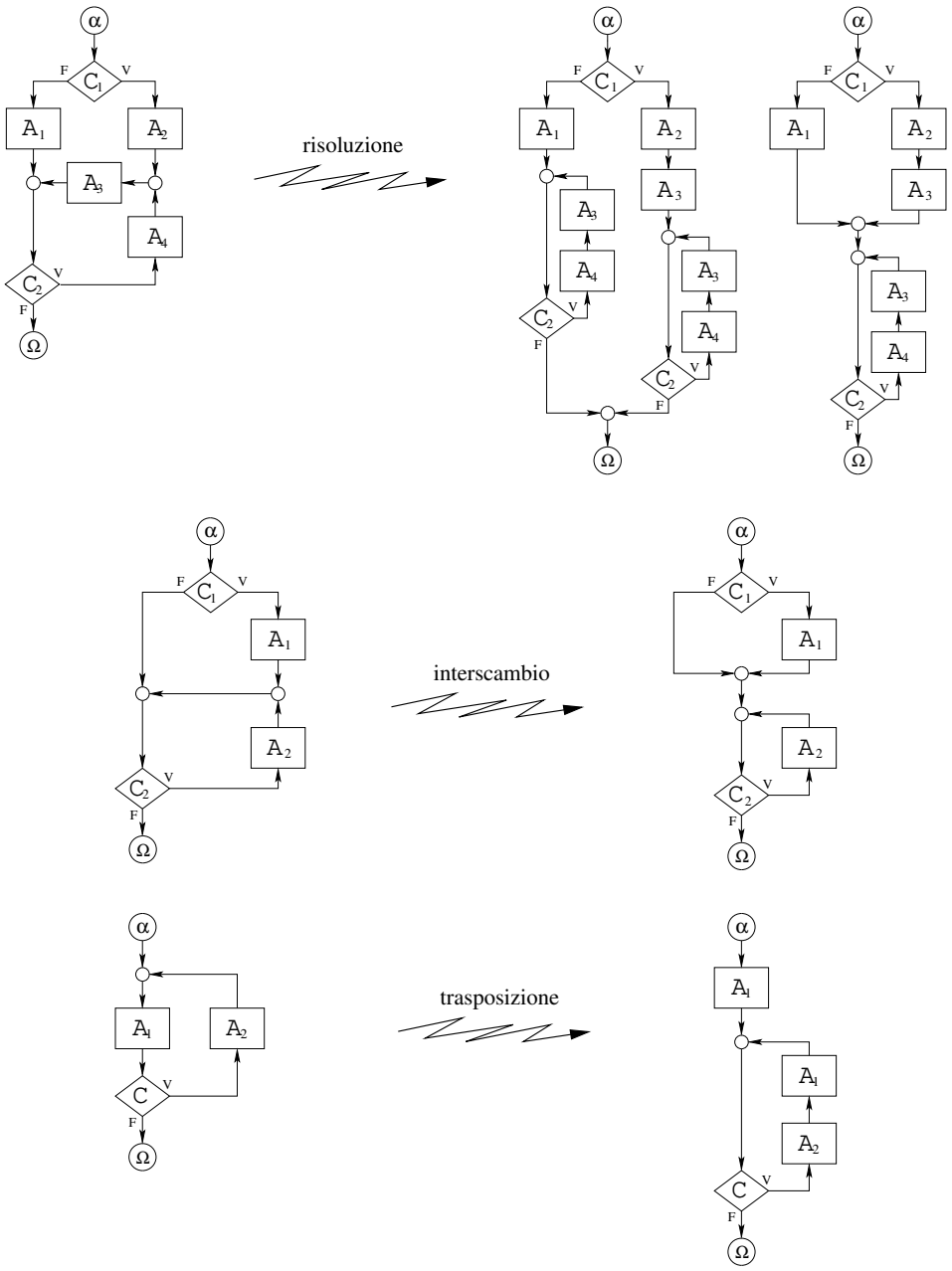


Figura 4.2: Schemi di flusso strutturati

- Proprietà: Condizione necessaria affinché uno schema di flusso sia strutturato è che esso abbia un unico nodo di inizio e un unico nodo di fine.
- Teorema fondamentale della programmazione strutturata (Böhm e Jacopini): Dato un programma P e uno schema di flusso F che lo descrive, è sempre possibile determinare un programma P' *equivalente* a P che è descrivibile con uno schema di flusso F' *strutturato*.
- In virtù del teorema precedente, è sempre possibile evitare l'uso dell'istruzione `goto` qualora il linguaggio impiegato metta a disposizione dei meccanismi di sequenza, selezione e ripetizione di istruzioni, come avviene in tutti i moderni linguaggi di programmazione.
- Se uno schema di flusso non è strutturato, è possibile renderlo tale applicando le seguenti regole:
  - Risoluzione: duplicare e separare blocchi condivisi aggiungendo ulteriori nodi collettori.
  - Interscambio: scambiare le linee di ingresso/uscita di nodi collettori adiacenti.
  - Trasposizione: scambiare un blocco di azione con un nodo collettore o un blocco di controllo, a patto di preservare la semantica.

- Esempi di applicazione delle regole di strutturazione:



Nel primo esempio, ci sono due alternative a seconda che si separino le due vie dell'istruzione di selezione iniziale (duplicando l'istruzione di ripetizione finale) oppure l'istruzione di selezione iniziale dall'istruzione di ripetizione finale (duplicando soltanto il blocco di azione `A3`). La seconda alternativa è dunque preferibile perché comporta meno ridondanza della prima alternativa. ■fltpp_9

Capitolo 5

Procedure

5.1 Formato di un Programma con Più Funzioni su un Singolo File

- Un problema complesso viene di solito affrontato suddividendolo in sottoproblemi più semplici. Ciò è supportato dal paradigma di programmazione imperativo di natura procedurale attraverso la possibilità di articolare il programma che dovrà risolvere il problema in più sottoprogrammi o procedure – detti funzioni nel linguaggio C – che dovranno risolvere i vari sottoproblemi. Tale meccanismo prende il nome di progettazione top-down e consente lo sviluppo di programmi modulari per raffinamenti successivi.
- Vantaggi derivanti dalla suddivisione di un programma C in funzioni:
  - Maggiore leggibilità del programma conseguente allo snellimento della funzione `main`.
  - Possibilità di suddividere il lavoro in modo coordinato all'interno di un gruppo di programmatori.
  - Riuso del software: una funzione viene definita una sola volta, ma può essere usata più volte sia all'interno del programma in cui è definita che in altri programmi (minore lunghezza dei programmi, minore tempo necessario per scrivere i programmi, maggiore affidabilità dei programmi).
- Una funzione C è una sequenza di istruzioni logicamente correlate che lavorano su un insieme di dati parametrizzati. È utile esprimere tale sequenza di istruzioni come funzione quando la loro esecuzione è ripetuta in punti diversi del programma (se la loro esecuzione fosse ripetuta in un solo punto, basterebbe inserirle all'interno di un'istruzione di ripetizione).
- Una funzione C è assimilabile a una funzione matematica $f : A \to B$, $f(a) = b$, dove diciamo che f è l'identificatore della funzione, A è il tipo dei parametri, B è il tipo del risultato, $f : A \to B$ è la dichiarazione della funzione ed $f(a) = b$ è la definizione della funzione.
- Formato di un programma C con più funzioni su un singolo file:

  ◁*direttive al preprocessore*▷
  ◁*definizione dei tipi*▷
  ◁*dichiarazione delle variabili globali*▷
  ◁*dichiarazione delle funzioni (esclusa* `main`*)*▷
  ◁*definizione delle funzioni (a partire da* `main` *per coerenza con ordine top-down)*▷
- L'ordine in cui le funzioni vengono dichiarate o definite è inessenziale dal punto di vista dell'esecuzione del programma (l'ordine di dichiarazione e l'ordine di definizione devono coincidere). È invece importante che le funzioni vengano tutte dichiarate prima di essere definite, altrimenti il compilatore emette messaggi d'errore in caso di invocazioni incrociate tra più funzioni (se `f` invoca `g` e `g` invoca `f` il compilatore segnala che la prima funzione invocata non è stata definita).
- L'esecuzione del programma è sequenziale a meno di chiamate di funzione. Essa inizia sempre dalla prima istruzione della funzione `main` e poi continua seguendo l'ordine testuale delle istruzioni presenti nella funzione `main` e nelle funzioni che vengono via via invocate.
- Le variabili globali sono utilizzabili all'interno di ciascuna funzione che segue la loro dichiarazione. Come nel caso dell'istruzione `goto`, l'uso delle variabili globali è sconsigliato perché il fatto che più funzioni possano modificare il valore di tali variabili rende difficile tenere sotto controllo l'evoluzione dei valori che le variabili stesse assumono a tempo di esecuzione.

5.2 Dichiarazione di Funzione

- La sintassi della dichiarazione di una funzione C è la seguente:
  ◁*tipo risultato*▷ ◁*identificatore funzione*▷(◁*tipi parametri formali*▷);
- Il tipo del risultato rappresenta il tipo del valore che viene restituito alla funzione chiamante quando l'esecuzione della funzione termina. Tale tipo è `void` se la funzione non restituisce alcun risultato.
- I tipi dei parametri formali sono costituiti dalla sequenza dei tipi degli argomenti della funzione separati da virgole. Tale sequenza è `void` se la funzione non ha argomenti.

5.3 Definizione di Funzione e Parametri Formali

- La sintassi della definizione di una funzione C è la seguente:
  ◁*tipo risultato*▷ ◁*identificatore funzione*▷(◁*dichiarazione parametri formali*▷)
  {
  ◁*dichiarazione variabili locali*▷
  ◁*istruzioni della funzione*▷
  }
  dove la prima linea rappresenta l'intestazione della funzione, mentre ciò che è racchiuso tra parentesi graffe rappresenta il corpo della funzione ed è indentato per motivi di leggibilità.
- La dichiarazione dei parametri formali è costituita da una sequenza di dichiarazioni di variabili separate da virgole che rappresentano gli argomenti della funzione (`void` se la funzione non ha argomenti).
- L'intestazione della funzione deve coincidere con la dichiarazione della funzione a meno dei nomi dei parametri (necessari per far riferimento ai parametri nelle istruzioni) e del punto e virgola finale.
- Gli identificatori dei parametri formali e delle variabili locali sono utilizzabili solo all'interno del corpo della funzione.

5.4 Invocazione di Funzione e Parametri Effettivi

- La sintassi dell'invocazione di una funzione C è la seguente:
  ◁*identificatore funzione*▷(◁*parametri effettivi*▷)
  dove l'invocazione può formare un'intera istruzione, nel qual caso è terminata da un punto e virgola, oppure trovarsi all'interno di un'espressione.
- I parametri effettivi sono costituiti da una sequenza di espressioni separate da virgole, i cui valori sono usati ordinatamente da sinistra a destra per inizializzare i parametri formali della funzione invocata (la sequenza è vuota se la funzione invocata non ha argomenti).
- Parametri effettivi e parametri formali della funzione invocata devono corrispondere per numero, ordine e tipo. Se un parametro effettivo e il corrispondente parametro formale hanno tipi diversi compresi nell'insieme {`int`, `double`}, valgono le osservazioni fatte in Sez. 3.10 per gli operatori di assegnamento.
- Dal punto di vista dell'esecuzione delle istruzioni, l'effetto dell'invocazione di una funzione è quello di interrompere l'esecuzione della funzione chiamante e far diventare la prima istruzione della funzione chiamata la prossima istruzione da eseguire.

5.5 Istruzione `return`

- Se il tipo del risultato di una funzione è diverso da `void`, nel corpo della funzione sarà presente la seguente istruzione:
  `return`(◁*espressione*▷);
  la quale restituisce alla funzione chiamante come risultato della funzione considerata il valore dell'espressione, che deve essere del tipo dichiarato per il risultato della funzione.
- Dal punto di vista dell'esecuzione delle istruzioni, l'effetto dell'istruzione `return` è quello di terminare l'esecuzione della funzione chiamata e far diventare l'istruzione della funzione chiamante successiva a quella contenente l'invocazione della funzione la prossima istruzione da eseguire, riprendendo così l'esecuzione della funzione chiamante dal punto in cui era stata interrotta.
- Per coerenza con i principi della programmazione strutturata, una funzione che restituisce un risultato deve contenere un'unica istruzione `return` (evitare istruzioni di selezione contenenti molteplici `return`). Inoltre, nel corpo della funzione tale istruzione deve essere l'ultima, perché ulteriori istruzioni non potrebbero mai essere eseguite (codice morto).

5.6 Parametri e Risultato della Funzione `main`

- La funzione `main` è dotata di due parametri formali inizializzati dal sistema operativo in base alle stringhe (opzioni e nomi di file) presenti nel comando con cui il programma viene lanciato in esecuzione.
- Se tali parametri debbono essere utilizzabili all'interno del programma, l'intestazione della funzione `main` deve essere estesa come segue (notare l'allineamento dei parametri per motivi di leggibilità):
  ```
  int main(int   argc,
           char *argv[])
  ```
- Il parametro `argc` contiene il numero di stringhe presenti nel comando, incluso il nome del file eseguibile del programma.
- Il parametro `argv` è un vettore contenente le stringhe presenti nel comando, incluso il nome del file eseguibile del programma.
- Esempio di lancio in esecuzione di un programma il cui file eseguibile si chiama `pippo`:
  ```
  pippo -r dati.txt
  ```
  dove l'opzione specificata stabilisce se il file che la segue deve essere letto o scritto. In questo caso, `argc` vale `3` e `argv` contiene le stringhe `"pippo"`, `"-r"` e `"dati.txt"`.
- Il risultato restituito dalla funzione `main` attraverso l'istruzione `return` è un valore di controllo che viene passato al sistema operativo per verificare se l'esecuzione del programma è andata a buon fine (no terminazione prematura). Il valore restituito in caso di normale terminazione è `0`. ■fltpp_10

5.7 Passaggio di Parametri per Valore e per Indirizzo

- Nella dichiarazione di un parametro formale di una funzione occorre stabilire se il corrispondente parametro effettivo deve essere passato per valore o per indirizzo:
  - Se il parametro effettivo viene passato per valore, il valore della relativa espressione viene copiato nell'area di memoria riservata al corrispondente parametro formale.
  - Se il parametro effettivo viene passato per indirizzo, la relativa espressione viene interpretata come un indirizzo di memoria e questo viene copiato nell'area di memoria riservata al corrispondente parametro formale.
- Qualora il parametro effettivo sia una variabile, nel primo caso il valore della variabile non può essere modificato dalla funzione invocata durante la sua esecuzione, in quanto ciò che viene passato è una copia del valore di quella variabile. Per contro, nel secondo caso il corrispondente parametro formale contiene l'indirizzo della variabile, quindi attraverso questo parametro la funzione invocata può modificare durante la sua esecuzione il valore della variabile.
- Diversamente dal passaggio per valore, il passaggio per indirizzo deve essere esplicitamente dichiarato:
  - Se un parametro effettivo passato per indirizzo è di tipo *`tipo`*, il corrispondente parametro formale deve essere dichiarato di tipo *`tipo *`*.
  - Se `p` è un parametro formale di tipo *`tipo *`*, all'interno delle istruzioni della funzione in cui `p` è dichiarato si denota con `p` l'indirizzo contenuto in `p`, mentre si denota con `*p` il valore contenuto nell'area di memoria il cui indirizzo è contenuto in `p` (vedi operatore valore-di in Sez. 6.11).
  - Se `v` è una variabile passata per indirizzo, essa viene denotata con `&v` all'interno dell'invocazione di funzione (vedi operatore indirizzo-di in Sez. 6.11).
- Un parametro è di input, output o input/output a seconda che la funzione faccia uso del suo valore, calcoli il suo valore o entrambe le cose. Normalmente i parametri di una funzione sono dati di input, nel qual caso il passaggio per valore è sufficiente. Se però in una funzione alcuni parametri rappresentano dati di input/output, oppure la funzione – come nel caso della `scanf` – deve restituire più risultati (l'istruzione `return` permette di restituirne uno solo), allora è necessario il passaggio per indirizzo.

- Esempio di differenza tra passaggio per valore e passaggio per indirizzo di una variabile:

```
...
  pippo1(v);
  w = v + 3;
...
void pippo1(int n)
{
  n += 10;
  printf("valore incrementato: %d\n",
         n);
}
...
```

```
...
  pippo2(&v);
  w = v + 3;
...
void pippo2(int *n)
{
  *n += 10;
  printf("valore incrementato: %d\n",
         *n);
}
...
```

  Se `v` ha valore `5`, in entrambi i casi il valore che viene stampato è `15`. La differenza è che nel primo caso il valore che viene assegnato a `w` è `8`, mentre nel secondo caso è `18` perché `v` diventa `15`.

- Esempio di programma: aritmetica con le frazioni.

  1. **Specifica del problema**. Calcolare il risultato della addizione, sottrazione, moltiplicazione o divisione di due frazioni, mostrandolo ancora in forma di frazione.
  2. **Analisi del problema**. I dati di ingresso del problema sono rappresentati dalle due frazioni e dall'operatore aritmetico da applicare ad esse. I dati di uscita del problema sono rappresentati dal risultato dell'applicazione dell'operatore aritmetico alle due frazioni, con il risultato da esprimere ancora sotto forma di frazione. Le relazioni da sfruttare sono le leggi dell'aritmetica con le frazioni: date $\frac{n_1}{d_1}$ ed $\frac{n_2}{d_2}$ dove $n_1, n_2 \in \mathbb{Z}$ e $d_1, d_2 \in \mathbb{Z} \setminus \{0\}$, vale che $\frac{n_1}{d_1} \pm \frac{n_2}{d_2} = \frac{n_1 \cdot d_2 \pm n_2 \cdot d_1}{d_1 \cdot d_2}$, $\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 \cdot n_2}{d_1 \cdot d_2}$, $\frac{n_1}{d_1} : \frac{n_2}{d_2} = \frac{n_1}{d_1} \cdot \frac{d_2}{n_2}$ (se $n_2 \neq 0$).
  3. **Progettazione dell'algoritmo**. Poiché una frazione è costituita da due numeri interi detti rispettivamente numeratore e denominatore, dove il denominatore deve essere diverso da zero, conveniamo di rappresentare il segno della frazione nel numeratore, cosicché il denominatore deve essere un numero intero strettamente positivo. Stabiliamo inoltre di rappresentare l'espressione in notazione infissa, cosicché l'operatore deve essere acquisito tra le due frazioni.
     I passi dell'algoritmo sono i seguenti:
     - Acquisire la prima frazione.
     - Acquisire l'operatore aritmetico.
     - Acquisire la seconda frazione.
     - Applicare l'operatore aritmetico.
     - Comunicare il risultato sotto forma di frazione.

     I passi riportati sopra possono essere svolti attraverso altrettante chiamate di funzioni. Si può pensare di sviluppare una funzione per la lettura di una frazione (che verrà richiamata due volte), una funzione per la lettura di un operatore aritmetico e una funzione per la stampa di una frazione. Si può inoltre pensare di sviluppare una funzione per ciascuna delle quattro operazioni aritmetiche. In realtà, abbiamo soltanto bisogno di una funzione per l'addizione – utilizzabile anche in una sottrazione a patto di cambiare preventivamente il segno del numeratore della seconda frazione – e di una funzione per la moltiplicazione – utilizzabile anche in una divisione a patto di scambiare preventivamente tra loro numeratore e denominatore della seconda frazione.

4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*********************************************/
/* programma per l'aritmetica con le frazioni */
/*********************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*********************************/
/* dichiarazione delle funzioni */
/*********************************/

void leggi_frazione(int *,
                    int *,
                    char *);
char leggi_operatore(void);
void somma_frazioni(int,
                    int,
                    int,
                    int,
                    int *,
                    int *);
void moltiplica_frazioni(int,
                         int,
                         int,
                         int,
                         int *,
                         int *);
void stampa_frazione(int,
                     int);

/*******************************/
/* definizione delle funzioni */
/*******************************/

/* definizione della funzione main */
int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  int  n1,   /* input: numeratore della prima frazione */
       d1,   /* input: denominatore della prima frazione */
       n2,   /* input: numeratore della seconda frazione */
       d2;   /* input: denominatore della seconda frazione */
  char op;   /* input: operatore aritmetico da applicare */
  int  n,    /* output: numeratore della frazione risultato */
       d;    /* output: denominatore della frazione risultato */

  /* acquisire la prima frazione */
  leggi_frazione(&n1,
                 &d1,
                 "prima");
```

```
  /* acquisire l'operatore aritmetico */
  op = leggi_operatore();

  /* acquisire la seconda frazione */
  leggi_frazione(&n2,
                 &d2,
                 "seconda");

  /* applicare l'operatore aritmetico */
  switch (op)
  {
    case '+':
      somma_frazioni(n1,
                     d1,
                     n2,
                     d2,
                     &n,
                     &d);
      break;
    case '-':
      somma_frazioni(n1,
                     d1,
                     -n2,
                     d2,
                     &n,
                     &d);
      break;
    case '*':
      moltiplica_frazioni(n1,
                          d1,
                          n2,
                          d2,
                          &n,
                          &d);
      break;
    case ':':
      if (n2 != 0)
        moltiplica_frazioni(n1,
                            d1,
                            d2,
                            n2,
                            &n,
                            &d);
      break;
  }

  /* comunicare il risultato sotto forma di frazione */
  if (op != ':' || n2 != 0)
    stampa_frazione(n,
                    d);
  else
    printf("Divisione illegale!\n");
  return(0);
}
```

```
/* definizione della funzione per leggere una frazione */
void leggi_frazione(int  *num,   /* output: numeratore della frazione */
                    int  *den,   /* output: denominatore della frazione */
                    char *msg)   /* input: messaggio specifico */
{
  /* dichiarazione delle variabili locali alla funzione */
  char sep;                  /* lavoro: carattere che separa numeratore e denominatore */
  int  esito_lettura,        /* lavoro: esito della scanf */
       acquisizione_errata;  /* lavoro: esito complessivo dell'acquisizione */
  /* leggere e validare la frazione */
  do
  {
    printf("Digita la %s frazione come coppia di interi separati da \"/\" ",
           msg);
    printf("con il secondo intero strettamente positivo: ");
    esito_lettura = scanf("%d %c%d",
                          num,
                          &sep,
                          den);
    acquisizione_errata = esito_lettura != 3 || sep != '/' || *den <= 0;
    if (acquisizione_errata)
      printf("Valori non accettabili!\n");
    while (getchar() != '\n');
  }
  while (acquisizione_errata);
}
/* definizione della funzione per leggere un operatore aritmetico */
char leggi_operatore(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  char op;   /* output: operatore aritmetico */
  /* leggere e validare l'operatore aritmetico */
  do
  {
    printf("Digita un operatore aritmetico (+, -, *, :): ");
    scanf(" %c",
          &op);
  }
  while (op != '+' && op != '-' && op != '*' && op != ':');
  return(op);
}
/* definizione della funzione per sommare due frazioni */
void somma_frazioni(int  n1,   /* input: numeratore della prima frazione */
                    int  d1,   /* input: denominatore della prima frazione */
                    int  n2,   /* input: numeratore della seconda frazione */
                    int  d2,   /* input: denominatore della seconda frazione */
                    int *n,    /* output: numeratore della frazione risultato */
                    int *d)    /* output: denominatore della frazione risultato */
{
  /* sommare le due frazioni */
  *n = n1 * d2 + n2 * d1;
  *d = d1 * d2;
}
```

```
/* definizione della funzione per moltiplicare due frazioni */
void moltiplica_frazioni(int  n1,   /* input: numeratore della prima frazione */
                         int  d1,   /* input: denominatore della prima frazione */
                         int  n2,   /* input: numeratore della seconda frazione */
                         int  d2,   /* input: denominatore della seconda frazione */
                         int *n,    /* output: numeratore della frazione risultato */
                         int *d)    /* output: denominatore della frazione risultato */
{
  /* moltiplicare le due frazioni */
  *n = n1 * n2;
  *d = d1 * d2;
}
/* definizione della funzione per stampare una frazione */
void stampa_frazione(int num,   /* input: numeratore della frazione */
                     int den)   /* input: denominatore della frazione */
{
  /* stampare la frazione */
  printf("La frazione risultato e' %d/%d\n",
         num,
         den);
}
```

■fltpp_11

5.8 Principio di Induzione e Funzioni Ricorsive

- Una funzione ricorsiva è una funzione che invoca direttamente o indirettamente se stessa. La ricorsione è adeguata per qualunque problema che sia suddivisibile in sottoproblemi più semplici che sono della *stessa natura di quello originario*, cioè per qualunque problema per il quale è possibile:
  - Individuare uno o più casi base, per i quali si può ricavare direttamente la soluzione del problema.
  - Definire uno o più casi generali attraverso un insieme di sottoproblemi della stessa natura di quello originario, che però sono più vicini ai casi base. La soluzione di un caso generale è data dalla combinazione delle soluzioni dei sottoproblemi tramite i quali il caso stesso è stato definito.
- La ricorsione è uno strumento molto potente perché consente di affrontare problemi altrimenti non gestibili (come il problema delle torri di Hanoi), risolvere certi problemi in maniera più efficiente (ad esempio gli algoritmi di ordinamento quicksort e mergesort) e definire in modo naturale determinate strutture dati (quali liste e alberi).
- Il fondamento teorico della ricorsione è il *principio di induzione*, cioè il quinto dei postulati introdotti alla fine del 1800 da Giuseppe Peano per dare una definizione assiomatica di $\mathbb{N}$ come il più piccolo insieme che contiene 0 ed è chiuso rispetto all'operazione unaria di successore:
  1. Esiste un elemento $0 \in \mathbb{N}$.
  2. Esiste una funzione $succ : \mathbb{N} \to \mathbb{N}$.
  3. Per ogni $n \in \mathbb{N}$, $succ(n) \neq 0$.
  4. Per ogni $n, n' \in \mathbb{N}$, se $n \neq n'$ allora $succ(n) \neq succ(n')$.
  5. Se M è un sottoinsieme di $\mathbb{N}$ tale che:
     (a) $0 \in M$;
     (b) per ogni $n \in \mathbb{N}$, $n \in M$ implica $succ(n) \in M$;
     allora $M = \mathbb{N}$.
- Gli elementi di $\mathbb{N}$ sono dunque 0, $succ(0)$, $succ(succ(0))$ e così via. Il principio di induzione consente altresì di definire *in modo finito* le quattro operazioni aritmetiche su $\mathbb{N}$ (cosiddetta aritmetica di Peano) avendo a disposizione la funzione $succ : \mathbb{N} \to \mathbb{N}$ e introducendo la funzione $pred : \mathbb{N}_{\neq 0} \to \mathbb{N}$ tale che $pred(succ(n)) = n$ per ogni $n \in \mathbb{N}$ e $succ(pred(n)) = n$ per ogni $n \in \mathbb{N}_{\neq 0}$. Le seguenti operazioni binarie, che hanno due parametri denotati con m ed n rispettivamente, sono tutte definite per induzione sul parametro n:

- Addizione: $m \oplus n = \begin{cases} m & \text{se } n = 0 \\ succ(m) \oplus pred(n) & \text{se } n \neq 0 \end{cases}$

  sulla cui base si definisce $m \leq n$ se e solo se esiste $m' \in \mathbb{N}$ tale che $m \oplus m' = n$ e di conseguenza $m < n$ se e solo se $m \leq n$ con $m \neq n$, nonché $m \geq n$ se e solo se $n \leq m$ ed $m > n$ se e solo se $n < m$.
- Sottrazione: $m \ominus n = \begin{cases} m & \text{se } n = 0 \\ pred(m) \ominus pred(n) & \text{se } n > 0 \end{cases}$ dove $m \geq n$.
- Moltiplicazione: $m \otimes n = \begin{cases} 0 & \text{se } n = 0 \\ m \oplus (m \otimes pred(n)) & \text{se } n > 0 \end{cases}$.
- Divisione: $m \oslash n = \begin{cases} 0 & \text{se } m < n \\ succ((m \ominus n) \oslash n) & \text{se } m \geq n \end{cases}$ dove $n \neq 0$.

- Esempi di applicazione delle precedenti definizioni:
  - $5 \oplus 2 = 6 \oplus 1 = 7 \oplus 0 = 7$.
  - $5 \ominus 2 = 4 \ominus 1 = 3 \ominus 0 = 3$.
  - $5 \otimes 2 = 5 \oplus (5 \otimes 1) = 5 \oplus (5 \oplus (5 \otimes 0)) = 5 \oplus (5 \oplus 0) = 5 \oplus 5 = \ldots = 10$.
  - $5 \oslash 2 = succ((5 \ominus 2) \oslash 2) = \ldots = succ(3 \oslash 2) = succ(succ((3 \ominus 2) \oslash 2)) = \ldots = succ(succ(1 \oslash 2)) = succ(succ(0)) = succ(1) = 2$.

- Il principio di induzione fornisce in generale un meccanismo per descrivere *in modo finito* un insieme *infinito numerabile* dotato di una relazione d'ordine totale che ammette l'elemento minimo. Questo meccanismo prende il nome di definizione ricorsiva e comprende uno o più casi base – in ciascuno dei quali la definizione è espressa in modo diretto – e uno o più casi induttivi – in ciascuno dei quali la definizione è espressa ricorrendo a una definizione della stessa natura che però è più vicina a uno dei casi base. Nella definizione ricorsiva bisogna specificare su quale entità si procede per induzione.

- Il principio di induzione fornisce inoltre una condizione sufficiente per verificare se una proprietà è soddisfatta da *tutti* gli elementi di un insieme *infinito numerabile* dotato di una relazione d'ordine totale che ammette l'elemento minimo. La dimostrazione si suddivide in due parti. Nella prima parte (caso base), si verifica direttamente se la proprietà è soddisfatta dall'elemento minimo dell'insieme. Nella seconda parte (caso induttivo), si verifica se la proprietà è soddisfatta da un generico elemento dell'insieme che è maggiore del minimo, assumendo che la proprietà sia soddisfatta dall'elemento che lo precede o da tutti gli elementi che lo precedono (ipotesi induttiva).

- Esempi:
  - In virtù dell'assiomatizzazione di Peano, le quattro operazioni aritmetiche su $\mathbb{N}$ possono essere espresse in modo ricorsivo utilizzando solo gli operatori di incremento e decremento di un'unità assieme agli operatori relazionali:

```
int addizione(int m,   /* input: m >= 0 */
              int n)   /* input: n >= 0 */
{
  int somma;   /* output: risultato */

  if (n == 0)
    somma = m;
  else
    somma = addizione(m + 1,
                      n - 1);
  return(somma);
}
```

```
int sottrazione(int m,   /* input: m >= n */
                int n)   /* input: n >= 0 */
{
  int differenza;   /* output: risultato */

  if (n == 0)
    differenza = m;
  else
    differenza = sottrazione(m - 1,
                             n - 1);
  return(differenza);
}

int moltiplicazione(int m,   /* input: m >= 0 */
                    int n)   /* input: n >= 0 */
{
  int prodotto;   /* output: risultato */

  if (n == 0)
    prodotto = 0;
  else
    prodotto = addizione(m,
                         moltiplicazione(m,
                                         n - 1));
  return(prodotto);
}

void divisione(int  m,           /* input: m >= 0 */
               int  n,           /* input: n > 0 */
               int *quoziente,   /* output: quoziente */
               int *resto)       /* output: resto */
{
  if (m < n)
  {
    *quoziente = 0;
    *resto = m;
  }
  else
  {
    divisione(sottrazione(m,
                          n),
              n,
              quoziente,
              resto);
    *quoziente += 1;
  }
}
```

– Altre operazioni matematiche su $\mathbb{N}$ possono essere espresse in modo ricorsivo utilizzando solo le quattro operazioni aritmetiche e gli operatori relazionali:

```
int elevamento(int m,   /* input: m >= 0 */
               int n)   /* input: n >= 0, n != 0 se m == 0 */
{
  int potenza;   /* output: risultato */

  if (n == 0)
    potenza = 1;
  else
    potenza = m * elevamento(m,
                             n - 1);
  return(potenza);
}

int massimo_comun_divisore(int m,   /* input: m >= n */
                           int n)   /* input: n > 0 */
{
  int mcd;   /* output: risultato */

  if (m % n == 0)
    mcd = n;
  else
    mcd = massimo_comun_divisore(n,
                                 m % n);
  return(mcd);
}

int fattoriale(int n)   /* input: n >= 0 */
{
  int fatt;   /* output: risultato */

  if (n == 0)                       /* for (i = fatt = 1; */
    fatt = 1;                       /*      (i <= n);     */
  else                              /*      i++)          */
    fatt = n * fattoriale(n - 1);   /*   fatt *= i;       */
  return(fatt);
}
```

– L'n-esimo numero di Fibonacci fib_n è il numero di coppie di conigli esistenti nel periodo n sotto le seguenti ipotesi: nel periodo 1 viene a esistere la prima coppia di conigli, nessuna coppia è fertile nel primo periodo successivo al periodo in cui è avvenuta la sua nascita, ogni coppia produce un'ulteriore coppia in ciascuno degli altri periodi successivi. Nel periodo 1 abbiamo dunque una sola coppia di conigli, che non è ancora fertile nel periodo 2 e comincia a produrre nuove coppie a partire dal periodo 3. Il numero di coppie esistenti nel generico periodo n è il numero di coppie esistenti nel periodo $n-1$ più il numero di nuove coppie nate nel periodo n, le quali sono tante quante le coppie fertili nel periodo n, che coincidono a loro volta con le coppie esistenti nel periodo $n-2$:

```
int fibonacci(int n)   /* input: n >= 1 */
{
  int fib;   /* output: risultato */

  if (n == 1 || n == 2)
    fib = 1;
  else
    fib = fibonacci(n - 1) + fibonacci(n - 2);
  return(fib);
}
```

– Il problema delle torri di Hanoi è il seguente. Date tre aste A, B, C di altezza sufficiente con n dischi diversi accatastati sull'asta A in ordine di diametro decrescente dal basso verso l'alto, portare i dischi sull'asta C rispettando le due seguenti regole: (i) è possibile spostare un solo disco alla volta e (ii) un disco non può mai essere appoggiato sopra un disco di diametro inferiore. Osservato che per $n = 1$ la soluzione è banale, quando $n \geq 2$ si può adottare un meccanismo ricorsivo del seguente tipo in cui i ruoli delle aste si scambiano. Spostare gli $n - 1$ dischi di diametro più piccolo dalla prima alla seconda asta usando questo meccanismo ricorsivo, poi spostare il disco di diametro più grande direttamente dalla prima alla terza asta, e infine spostare gli $n - 1$ dischi di diametro più piccolo dalla seconda alla terza asta usando questo meccanismo ricorsivo:

```
void hanoi(int  n,            /* input: n >= 1 */
           char partenza,     /* input: 'A' nella chiamata iniziale */
           char arrivo,       /* input: 'C' nella chiamata iniziale */
           char intermedia)   /* input: 'B' nella chiamata iniziale */
{
  if (n == 1)
    printf("Sposta da %c a %c.\n",
           partenza,
           arrivo);
  else
  {
    hanoi(n - 1,
          partenza,
          intermedia,
          arrivo);
    printf("Sposta da %c a %c.\n",
           partenza,
           arrivo);
    hanoi(n - 1,
          intermedia,
          arrivo,
          partenza);
  }
}
```

■fltpp_12

5.9 Modello di Esecuzione Sequenziale Basato su Pila

- Le istruzioni di un programma C vengono eseguite una alla volta nell'ordine in cui sono state scritte e la loro esecuzione è sequenziale a meno di chiamate di funzione. Quando il programma viene lanciato in esecuzione, il sistema operativo riserva tre aree distinte di memoria principale per il programma:
  – Un'area per contenere la versione eseguibile delle istruzioni del programma.
  – Un'area destinata come stack (o pila) per contenere un record di attivazione per ogni invocazione di funzione la cui esecuzione non è ancora terminata.
  – Un'area destinata come heap per l'allocazione/disallocazione delle strutture dati dinamiche.
- A seguito dell'invocazione di una funzione, il sistema operativo compie i seguenti passi (in maniera del tutto trasparente all'utente del programma):
  – Un record di attivazione per la funzione viene allocato in cima allo stack, la cui dimensione è tale da poter contenere i valori di parametri formali e variabili locali della funzione, più alcune informazioni di controllo. Questo record di attivazione viene a trovarsi subito sopra a quello della funzione che ha invocato la funzione in esame.
  – Lo spazio riservato ai parametri formali viene inizializzato con i valori dei corrispondenti parametri effettivi contenuti nell'invocazione, mentre quello riservato alle variabili locali non viene inizializzato con valori specifici.

  - Tra le informazioni di controllo viene memorizzato l'indirizzo dell'istruzione eseguibile successiva a quella contenente l'invocazione (indirizzo di ritorno), il quale viene preso dal program counter.
  - Il registro program counter viene impostato con l'indirizzo della prima istruzione eseguibile della funzione invocata, così da continuare l'esecuzione del programma da quella istruzione.

- A seguito della terminazione dell'esecuzione di una funzione, il sistema operativo compie i seguenti passi (in maniera del tutto trasparente all'utente del programma):
  - Il record di attivazione viene disallocato dalla cima dello stack, cosicché il record di attivazione della funzione che ha invocato la funzione in esame viene di nuovo a trovarsi in cima allo stack.
  - L'eventuale risultato restituito dalla funzione viene memorizzato nel record di attivazione della funzione che ha invocato la funzione in esame.
  - Il registro program counter viene impostato con l'indirizzo di ritorno precedentemente memorizzato nel record di attivazione, così da riprendere l'esecuzione del programma da quel punto.

- Invece di allocare staticamente un record di attivazione per ogni funzione all'inizio dell'esecuzione del programma, si utilizza un modello di esecuzione a pila (implementato attraverso lo stack dei record di attivazione) in cui i record di attivazione sono associati alle invocazioni delle funzioni – non alle funzioni – e vengono dinamicamente allocati/disallocati in ordine last-in-first-out (LIFO).

- Il motivo per cui si usa il modello di esecuzione a pila anziché il più semplice modello statico è che quest'ultimo non supporta la corretta esecuzione delle funzioni ricorsive. Prevedendo un unico record di attivazione per ciascuna funzione ricorsiva (invece di un record distinto per ogni invocazione di una funzione ricorsiva), il modello statico provoca interferenze tra le diverse invocazioni ricorsive di una funzione. Infatti in tale modello ogni invocazione ricorsiva finisce per sovrascrivere dentro al record di attivazione della funzione i valori dei parametri formali e delle variabili locali della precedente invocazione ricorsiva, determinando così il calcolo di un risultato errato.

5.10 Formato di un Programma con Più Funzioni su Più File

- Un programma C articolato in funzioni può essere distribuito su più file. Questa organizzazione, ormai prassi consolidata nel caso di grossi sistemi software, si basa sullo sviluppo e sull'utilizzo di librerie, ciascuna delle quali contiene funzioni e strutture dati logicamente correlate tra loro.
- L'organizzazione di un programma C complesso su più file enfatizza i vantaggi dell'articolazione del programma in funzioni:
  - Le funzionalità offerte dalle funzioni e dalle strutture dati di una libreria ("cosa") possono essere separate dai relativi dettagli implementativi ("come"), che rimangono di conseguenza nascosti a chi utilizza la libreria e possono essere modificati in ogni momento senza alterare le funzionalità offerte dalla libreria stessa.
  - Il grado di riuso del software aumenta, in quanto una funzione o una struttura dati di una libreria può essere usata più volte non solo all'interno di un unico programma, ma all'interno di tutti i programmi che includeranno la libreria (in generale, sviluppare una libreria ha senso solo se è ragionevole prevederne l'uso in più programmi).

- Una libreria C consiste in un file di implementazione e un file di intestazione aventi nomi coerenti tra loro:
  - Il file di implementazione (`.c`) ha il seguente formato:

    ◁*direttive al preprocessore (costanti da esportare e interne)*▷
    ◁*definizione dei tipi (da esportare e interni)*▷
    ◁*dichiarazione delle variabili globali (da esportare e interne)*▷
    ◁*dichiarazione delle funzioni (da esportare e interne)*▷
    ◁*definizione delle funzioni (da esportare e interne) (no* `main`*)*▷

In altri termini, il formato è lo stesso di un programma con più funzioni su un singolo file, ad eccezione della funzione `main` che non può essere definita all'interno di una libreria. Viene inoltre fatta distinzione tra gli identificatori da esportare – cioè utilizzabili nei programmi che includeranno il file di intestazione della libreria – e gli identificatori interni alla libreria – cioè la cui definizione è di supporto alla definizione degli identificatori da esportare.

– Il file di intestazione (`.h`) ha il seguente formato:
  ◁*ridefinizione delle costanti esportate*▷
  ◁*ridefinizione dei tipi esportati*▷
  ◁*ridichiarazione delle variabili globali esportate (precedute da* `extern`*)*▷
  ◁*ridichiarazione delle funzioni esportate (precedute da* `extern`*)*▷
  Questo file di intestazione esporta, cioè rende disponibili ai programmi che includeranno il file di intestazione stesso, tutti gli identificatori in esso contenuti. La ridichiarazione delle variabili globali e delle funzioni esportate deve essere preceduta da `extern`. Il file di intestazione non deve essere incluso nel corrispondente file di implementazione, perché quest'ultimo già contiene le definizioni e dichiarazioni (non precedute da `extern`) degli identificatori da esportare.

- In un programma organizzato su più file esiste solitamente un modulo principale – che è un file `.c` – il quale contiene la definizione della funzione `main` – che deve essere unica in tutto il programma – e include i file di intestazione di tutte le librerie necessarie. Il modulo principale e i file di implementazione delle librerie incluse vengono compilati separatamente in modo parziale, producendo così altrettanti file oggetto che vengono poi collegati assieme per ottenere un unico file eseguibile.

- Durante la compilazione parziale del modulo principale, il fatto che le ridichiarazioni delle funzioni importate dal modulo stesso siano precedute da `extern` consente al compilatore di sapere che i relativi identificatori sono definiti altrove, così da rimandare la ricerca delle loro definizioni al passo di linking. Se la ridichiarazione di una funzione importata dal modulo principale non fosse preceduta da `extern`, il compilatore cercherebbe la definizione di quella funzione nel modulo principale e, non trovandola, segnalerebbe errore.

- Esempio di libreria: aritmetica con le frazioni.

  – Il file di implementazione `frazioni.c` è uguale a quello riportato nella Sez. 5.7 dopo aver tolto la definizione della funzione `main`, quindi contiene tutte le dichiarazioni e definizioni di funzioni.

  – Il file di intestazione `frazioni.h`, da non includere nel file di implementazione `frazioni.c`, è il seguente:

```
/*****************************************************************/
/* intestazione della libreria per l'aritmetica con le frazioni */
/*****************************************************************/

/*******************************************/
/* ridichiarazione delle funzioni esportate */
/*******************************************/

extern void leggi_frazione(int *,
                           int *,
                           char *);
extern char leggi_operatore(void);
extern void somma_frazioni(int,
                           int,
                           int,
                           int *,
                           int *);
```

```
extern void moltiplica_frazioni(int,
                                int,
                                int,
                                int,
                                int *,
                                int *);
extern void stampa_frazione(int,
                            int);
```

– Il modulo principale (`.c`) contiene `#include "frazioni.h"`, così da poter invocare le funzioni esportate dalla libreria, e la definizione della funzione `main` della Sez. 5.7.

5.11 Visibilità degli Identificatori Locali e Non Locali

- A ogni identificatore presente in un programma C è associato un campo di visibilità, che definisce la regione del programma in cui l'identificatore è utilizzabile. Questa regione varia a seconda che l'identificatore sia locale o non locale.
- Un identificatore locale denota un parametro formale o una variabile locale di una funzione e ha come campo di visibilità soltanto la funzione stessa. All'inizio dell'esecuzione di un'invocazione della funzione, ogni parametro formale è inizializzato col valore del corrispondente parametro effettivo contenuto nell'invocazione, mentre il valore di ciascuna variabile locale è indefinito a meno che la variabile non sia esplicitamente inizializzata nella sua dichiarazione. Gli identificatori dei parametri formali e delle variabili locali di una funzione devono essere tutti distinti e sono memorizzati nel record di attivazione di ogni invocazione di quella funzione.
- Un identificatore non locale denota una costante simbolica, un tipo, una variabile globale o una funzione e ha come campo di visibilità la parte del file di implementazione in cui l'identificatore è definito/dichiarato compresa tra la sua definizione/dichiarazione e il termine del file, escluse quelle funzioni in cui viene dichiarato un parametro formale o una variabile locale con lo stesso nome di quell'identificatore (da evitare perché in tal caso in quelle funzioni si perde la possibilità di usare l'identificatore col suo significato originale). Gli identificatori non locali di un programma devono essere tutti distinti.
- Esistono inoltre i seguenti qualificatori per modificare il campo di visibilità di identificatori non locali:
  - Se `static` precede la dichiarazione di una variabile globale o di una funzione in un file `.c`, il relativo identificatore non può essere esportato al di fuori di quel file di implementazione (utile nei file di implementazione delle librerie per impedire di esportare identificatori la cui definizione è solo di supporto agli identificatori da esportare). In altri termini, `static` congela il campo di visibilità dell'identificatore di una variabile globale o di una funzione limitandolo al file di implementazione nel quale l'identificatore è definito.
  - Se `extern` precede la ridichiarazione di una variabile globale o di una funzione in un file `.h`, il relativo identificatore è definito in un file di implementazione diverso da quello che include quel file di intestazione. In altri termini, `extern` permette di ampliare il campo di visibilità dell'identificatore di una variabile globale o di una funzione, rendendo l'identificatore visibile al di fuori del file di implementazione nel quale è definito (fondamentale per poter attuare l'esportazione di identificatori attraverso i file di intestazione delle librerie).
- Esistono infine i seguenti qualificatori per modificare la memorizzazione di identificatori locali:
  - Se `static` precede la dichiarazione di una variabile locale, la variabile locale viene allocata una volta per tutte all'inizio dell'esecuzione del programma, invece di essere allocata in cima allo stack dei record di attivazione a ogni invocazione della relativa funzione. Ciò consente alla variabile di mantenere il valore che essa aveva al termine dell'esecuzione dell'invocazione precedente quando inizia l'esecuzione dell'invocazione successiva della relativa funzione (utile per alcune applicazioni, come i generatori di numeri pseudo-casuali, per evitare il ricorso a variabili globali).

– Se `register` precede la dichiarazione di un parametro formale o di una variabile locale, il parametro formale o la variabile locale viene allocato, se possibile, in un registro della CPU anziché in una cella di memoria principale (utile per fare accesso più rapidamente ai parametri formali e alle variabili locali più frequentemente utilizzate). ■fltpp_13

Capitolo 6

Tipi di Dati

6.1 Classificazione dei Tipi di Dati e Operatore `sizeof`

- Un tipo di dato denota, come se fosse una struttura algebrica, un insieme di valori a cui sono applicabili solo determinate operazioni. L'attribuzione di un tipo a ogni identificatore presente in un programma consente al compilatore di rilevare errori staticamente, cioè senza eseguire il programma.
- In generale i tipi di dati si suddividono in:
  - Tipi scalari che denotano insiemi di valori scalari, cioè non ulteriormente strutturati al loro interno (come i numeri, i valori di verità e i caratteri).
  - Tipi strutturati che denotano invece insiemi di valori aggregati i cui elementi possono essere:
    - omogenei, cioè dello stesso tipo, come nel caso di vettori e stringhe;
    - eterogenei, cioè di tipi eventualmente diversi, come nel caso di record e di strutture lineari, gerarchiche e reticolari di dimensione dinamicamente variabile (liste, alberi e grafi).
- In relazione ai tipi di dati del linguaggio C, si fa distinzione tra:
  - Tipi scalari predefiniti, che sono `int` (e le sue varianti), `double` (e le sue varianti) e `char`.
  - Tipi standard, che sono definiti nelle librerie standard (p.e. `FILE`).
  - Costruttori di tipo, che sono:
    - `enum` e puntatore ("`*`") per i tipi scalari;
    - array ("`[]`") e `struct` / `union` per i tipi strutturati omogenei ed eterogenei rispettivamente.
- Nuovi tipi di dati possono essere definiti in un programma C mediante la seguente sintassi:
  `typedef` ◁*definizione del tipo*▷ ◁*identificatore del tipo*▷`;`
  usando in *definizione del tipo* i tipi scalari predefiniti, i tipi standard e i costruttori di tipo.
- L'informazione sulla quantità di memoria in byte necessaria per rappresentare un valore di un certo tipo, che dipende dalla specifica implementazione del linguaggio C, è reperibile tramite l'operatore:
  `sizeof(`◁*tipo*▷`)`

6.2 Tipo `int`: Rappresentazione e Varianti

- Il tipo `int` denota l'insieme dei numeri interi rappresentabili in memoria con un certo numero di bit (sottoinsieme finito di $\mathbb{Z}$).
- Ogni valore numerico intero presente in un programma C viene espresso in base 10 nel file sorgente e rappresentato in base 2 in memoria principale a tempo d'esecuzione.
- Il minimo (risp. massimo) numero intero rappresentabile è indicato dalla costante simbolica `INT_MIN` (risp. `INT_MAX`) definita nel file di intestazione di libreria standard `limits.h`. Se si esce dalla gamma di valori `INT_MIN` .. `INT_MAX`, si ha un errore di overflow con generazione del valore `NaN` (not a number).
- Il numero di bit usati per la rappresentazione in memoria di un numero intero è dato da $\log_2$ `INT_MAX`, più un bit per la rappresentazione del segno.

- Varianti del tipo `int` e relative gamme minime di valori stabilite dallo standard ANSI, con indicazione del numero di bit usati per la loro rappresentazione in memoria:

`int`	-32767 .. 32767	1 + 15 bit
`unsigned`	0 .. 65535	16 bit
`short`	-32767 .. 32767	1 + 15 bit
`unsigned short`	0 .. 65535	16 bit
`long`	-2147483647 .. 2147483647	1 + 31 bit
`unsigned long`	0 .. 4294967295	32 bit

- Per la variante `unsigned` esiste il segnaposto `%u` ma conviene usare `%d` perché, se in corrispondenza di `%u` si digita un intero negativo, questo viene trasformato in un intero positivo tramite complementazione numerica e quindi per la validazione non si può fare affidamento sul risultato restituito da `scanf`.

6.3 Tipo `double`: Rappresentazione e Varianti

- Il tipo `double` denota l'insieme dei numeri reali rappresentabili in memoria con un certo numero di bit (sottoinsieme finito di $\mathbb{R}$).
- Ogni valore numerico reale presente in un programma C viene espresso in base 10 nel file sorgente (virgola fissa o virgola mobile) e rappresentato con due numeri interi in base 2 in memoria principale a tempo d'esecuzione (virgola mobile).
- Ogni numero reale (r) è rappresentato in memoria nel formato in virgola mobile attraverso due interi binari rispettivamente detti mantissa (m) ed esponente (e) tali che:
$$r = m \cdot 2^e$$
Il numero di bit riservati alla mantissa determina la precisione della rappresentazione, mentre il numero di bit riservati all'esponente determina l'ordine di grandezza della rappresentazione.
- Il minimo (risp. massimo) numero reale rappresentabile è indicato dalla costante simbolica `DBL_MIN` (risp. `DBL_MAX`) definita nel file di intestazione di libreria standard `float.h`. Se si esce dalla gamma di valori `DBL_MIN` .. `DBL_MAX`, si ha un errore di overflow con generazione del valore `NaN` (not a number).
- Il numero limitato di bit, ovvero concettualmente il numero limitato di cifre rappresentabili, può provocare anche errori di arrotondamento. In particolare, qualora un numero reale il cui valore assoluto è compreso tra 0 e 1 venga rappresentato come 0, si ha un errore di underflow.
- Il numero di bit usati per la rappresentazione in memoria di un numero reale è dato da $\log_2 \textit{mantissa}(\texttt{DBL\_MAX}) + \log_2 \textit{esponente}(\texttt{DBL\_MAX})$, più un bit per la rappresentazione del segno della mantissa e un bit per la rappresentazione del segno dell'esponente.
- Varianti del tipo `double` e relative gamme minime di valori positivi stabilite dallo standard ANSI, con indicazione del numero di bit usati per la rappresentazione in memoria dell'esponente:

`double`	10^{-307} .. 10^{308}	(2^{1024})	10 bit per l'esponente
`float`	10^{-37} .. 10^{38}	(2^{128})	7 bit per l'esponente
`long double`	10^{-4931} .. 10^{4932}	(2^{16384})	14 bit per l'esponente

6.4 Funzioni di Libreria Matematica

- Principali funzioni matematiche messe a disposizione dal linguaggio C, con indicazione dei relativi file di intestazione di libreria standard (spesso richiedono l'uso dell'opzione `-lm` nel comando di compilazione):

`int abs(int x)`	`stdlib.h`	$\lvert x \rvert$	
`double fabs(double x)`	`math.h`	$\lvert x \rvert$	
`double ceil(double x)`	`math.h`	$\lceil x \rceil$	
`double floor(double x)`	`math.h`	$\lfloor x \rfloor$	
`double sqrt(double x)`	`math.h`	$\sqrt{x}$	$x \geq 0$
`double exp(double x)`	`math.h`	e^x	
`double pow(double x, double y)`	`math.h`	x^y	$x < 0 \Rightarrow y \in \mathbb{Z}, x = 0 \Rightarrow y \neq 0$
`double log(double x)`	`math.h`	$\log_{\mathrm{e}} x$	$x > 0$
`double log10(double x)`	`math.h`	$\log_{10} x$	$x > 0$
`double sin(double x)`	`math.h`	$\sin x$	x espresso in radianti
`double cos(double x)`	`math.h`	$\cos x$	x espresso in radianti
`double tan(double x)`	`math.h`	$\tan x$	x espresso in radianti

6.5 Tipo char: Rappresentazione e Funzioni di Libreria

- Il tipo `char` denota l'insieme dei caratteri comprendente le 26 lettere minuscole, le 26 lettere maiuscole, le 10 cifre decimali, i simboli di punteggiatura, le parentesi, gli operatori aritmetici e relazionali e i caratteri di spaziatura (spazio, tabulazione, andata a capo).
- Ogni valore carattere presente in un programma C viene espresso racchiuso tra apici nel file sorgente e rappresentato con un numero intero in base 2 in memoria principale a tempo d'esecuzione.
- Ogni carattere è rappresentato in memoria attraverso una sequenza lunga solitamente 8 bit in conformità a un certo sistema di codifica, quale ASCII (American Standard Code for Information Interchange), EBCDIC (Extended Binary Coded Decimal Interchange Code di IBM), ecc.
- Per non limitare la portabilità di un programma C, ogni valore di tipo `char` usato nel programma deve essere espresso attraverso la relativa costante (p.e. `'A'`) anziché il rispettivo codice (p.e. `65` nel caso di ASCII) perché quest'ultimo potrebbe cambiare a seconda del sistema di codifica adottato nei computer su cui il programma verrà eseguito.
- Lo standard ANSI richiede che, qualunque sia il sistema di codifica adottato, esso garantisca che:
  - Le 26 lettere minuscole siano ordinatamente rappresentate attraverso 26 codici consecutivi.
  - Le 26 lettere maiuscole siano ordinatamente rappresentate attraverso 26 codici consecutivi.
  - Le 10 cifre decimali siano ordinatamente rappresentate attraverso 10 codici consecutivi.
- Poiché i caratteri sono codificati attraverso numeri interi, c'è piena compatibilità tra il tipo `char` e il tipo `int`. Ciò significa che variabili e valori di tipo `char` possono far parte di espressioni aritmetico-logiche.
- Esempi resi possibili dalla consecutività dei codici delle lettere minuscole, delle lettere maiuscole e delle cifre decimali:
  - Verifica del fatto che il carattere contenuto in una variabile di tipo `char` sia una lettera maiuscola:

```
char c;

if (c >= 'A' && c <= 'Z')   /* invece di: c == 'A' || c == 'B' || ... || c == 'Z' */
  ...
```

  - Trasformazione del carattere che denota una cifra decimale nel valore numerico corrispondente alla cifra stessa (se `'0'` è rappresentato da x allora `'1'` è rappresentato da $x + 1$ e così via):

```
char c;
int  n;

n = c - '0';
```

- Principali funzioni per il tipo `char` messe a disposizione dal linguaggio C, con indicazione dei relativi file di intestazione di libreria standard:

`int getchar(void)`	`stdio.h`	acquisisce un carattere da tastiera
`int putchar(int c)`	`stdio.h`	stampa un carattere su schermo
`int isalnum(int c)`	`ctype.h`	è un carattere alfanumerico?
`int isalpha(int c)`	`ctype.h`	è una lettera?
`int islower(int c)`	`ctype.h`	è una lettera minuscola?
`int isupper(int c)`	`ctype.h`	è una lettera maiuscola?
`int isdigit(int c)`	`ctype.h`	è una cifra decimale?
`int isspace(int c)`	`ctype.h`	è un carattere di spaziatura?
`int ispunct(int c)`	`ctype.h`	è un carattere diverso da lettera, cifra, spazio?
`int iscntrl(int c)`	`ctype.h`	è un carattere di controllo?
`int tolower(int c)`	`ctype.h`	trasforma una lettera in minuscolo
`int toupper(int c)`	`ctype.h`	trasforma una lettera in maiuscolo

6.6 Tipi Enumerati

- Nel linguaggio C è possibile costruire ulteriori tipi scalari mediante la seguente sintassi:
  `enum {◁`*identificatori dei valori*`▷}`
  dove è richiesto di enumerare esplicitamente gli identificatori dei valori assumibili dalle espressioni di questo tipo, con gli identificatori separati da virgole.
- Gli identificatori dei valori che compaiono nella definizione di un tipo enumerato devono essere diversi tra loro e non possono comparire nella definizione di un altro tipo enumerato.
- Se gli identificatori dei valori sono n, essi sono rappresentati da sinistra a destra mediante i numeri interi compresi tra 0 ed $n-1$. Ciò implica la piena compatibilità tra i tipi enumerati e il tipo `int`, quindi variabili e valori di un tipo enumerato possono far parte di espressioni aritmetico-logiche.
- Gli identificatori dei valori di un tipo enumerato sono assimilabili a costanti simboliche e perciò sono più comprensibili dell'uso diretto dei numeri.
- Esempi:
  - Definizione di un tipo per i valori di verità compatibile col fatto che zero rappresenta falso in C:

```
typedef enum {falso,
              vero}  booleano_t;
```

  - Definizione di un tipo per i giorni della settimana:

```
typedef enum {lunedi,
              martedi,
              mercoledi,
              giovedi,
              venerdi,
              sabato,
              domenica}  giorno_t;
```

  - Definizione di un tipo per i mesi dell'anno:

```
typedef enum {gennaio,
              febbraio,
              marzo,
              aprile,
              maggio,
              giugno,
              luglio,
              agosto,
              settembre,
              ottobre,
              novembre,
              dicembre}  mese_t;
```

  - Assegnamento di un valore enumerato (più leggibile rispetto all'uso di numeri):

```
giorno_t giorno;

giorno = lunedi;
```

  - Calcolo del giorno successivo (una volta arrivati a `domenica` bisogna ritornare a `lunedi`):

```
giorno_t oggi,
         domani;

domani = (oggi + 1) % 7;
```

6.7 Conversioni tra Tipi Scalari e Operatore di Cast

- I tipi scalari visti sinora e il tipo scalare puntatore (vedi Sez. 6.11) sono tra loro compatibili in qualche misura perché ogni loro valore è comunque riconducibile a un numero. Durante la valutazione delle espressioni aritmetico-logiche vengono effettuate le seguenti conversioni automatiche tra tali tipi, dove per `int` si intende anche un tipo ad esso assimilato come `char`, `enum` e puntatore:
  - Se un operatore binario è applicato a un operando di tipo `int` e un operando di tipo `double`, il valore dell'operando di tipo `int` viene convertito nel tipo `double` aggiungendogli una parte frazionaria nulla (`.0`) prima di applicare l'operatore.
  - Se un'espressione di tipo `int` deve essere assegnata a una variabile di tipo `double`, il valore dell'espressione viene convertito nel tipo `double` aggiungendogli una parte frazionaria nulla (`.0`) prima di essere assegnato alla variabile.
  - Se un'espressione di tipo `double` deve essere assegnata a una variabile di tipo `int`, il valore dell'espressione viene convertito nel tipo `int` tramite troncamento della parte frazionaria prima di essere assegnato alla variabile.
  - L'assegnamento dei valori dei parametri effettivi contenuti nell'invocazione di una funzione ai corrispondenti parametri formali della funzione invocata segue le regole precedenti.
- È inoltre possibile imporre delle conversioni esplicite di tipo alle espressioni di uno dei tipi scalari suddetti attraverso l'operatore di cast:
  (◁*tipo*▷)◁*espressione*▷
  Esso ha l'effetto di convertire nel tipo scalare specificato il valore dell'espressione cui è applicato prima che questo valore venga successivamente utilizzato. Se applicato a una variabile, l'operatore di cast non ne altera il contenuto e produce il suo effetto solo nel contesto dell'espressione in cui è applicato (cioè il tipo originariamente dichiarato per la variabile viene preservato).
- L'operatore di cast "`()`", che è unario e prefisso, ha la stessa precedenza degli operatori unari aritmetico-logici (vedi Sez. 3.11).
- Esempi:
  - Dato:

    ```
    double x;
    int    y,
           z;

    x = y / z;
    ```

    se `y` vale `3` e `z` vale `2`, il risultato della loro divisione è `1`, il quale viene automaticamente convertito in `1.0` prima di essere assegnato a `x`.
  - Dato:

    ```
    double x;
    int    y,
           z;

    x = (double)y / z;   /* oppure equivalentemente: x = y / (double)z; */
    ```

    se `y` vale `3` e `z` vale `2`, il valore di `y` viene esplicitamente convertito in `3.0` prima di effettuare la divisione, cosicché il valore assegnato a `x` è `1.5`.

6.8 Array: Rappresentazione e Operatore di Indicizzazione

- Il costruttore di tipo array del linguaggio C dà luogo a un valore aggregato formato da un numero finito di elementi dello stesso tipo, i quali sono memorizzati in celle consecutive di memoria.
- Una variabile di tipo array viene dichiarata come segue:
  ◁*tipo elementi*▷ ◁*identificatore variabile*▷[◁*espr_dich*▷];
  oppure con contestuale inizializzazione:
  ◁*tipo elementi*▷ ◁*identificatore variabile*▷[◁*espr_dich*▷] = {◁*sequenza valori*▷};
  dove:
  - Il numero di elementi (o lunghezza) della variabile di tipo array è dato da un'espressione di tipo `int` il cui valore deve essere positivo e i cui operandi devono essere delle costanti (no variabili). Ciò implica che il numero di elementi della variabile di tipo array è fissato staticamente.
  - Il numero di elementi può essere omesso se è specificata una sequenza di valori di inizializzazione separati da virgole aventi tutti tipo compatibile con quello dichiarato per gli elementi.
  - L'identificatore della variabile di tipo array rappresenta in forma simbolica l'indirizzo della locazione di memoria che contiene il valore del primo elemento dell'array.
- Essendo assimilabile a una costante simbolica, l'identificatore di una variabile di tipo array non può comparire in un'istruzione a sinistra di un operatore di assegnamento. Ciò implica in particolare che il risultato di una funzione non può essere di tipo array.
- Ogni elemento di una variabile di tipo array può essere selezionato all'interno di un'istruzione tramite il suo indice, che varia tra 0 ed $n-1$ se il numero di elementi della variabile è n. La sintassi è:
  ◁*identificatore variabile*▷[◁*espr_indice*▷]
  dove l'espressione deve essere di tipo `int` (variabili ammesse) e il suo valore deve essere compreso tra 0 ed $n-1$ altrimenti viene selezionato un elemento che sta al di fuori dello spazio di memoria riservato alla variabile di tipo array (salvo casi specifici, nessun messaggio d'errore viene emesso dal sistema operativo per segnalare tale situazione).
- In virtù della memorizzazione consecutiva degli elementi dell'array, del fatto che l'identificatore della variabile di tipo array rappresenta l'indirizzo del primo elemento e del fatto che l'indice di tale elemento è 0, l'indirizzo dell'elemento selezionato è ottenuto come segue:
  ◁*identificatore variabile*▷ + ◁*espr_indice*▷
- L'operatore di indicizzazione "`[]`", che è unario e postfisso, ha precedenza sugli operatori unari aritmetico-logici (vedi Sez. 3.11).
- Esempio di ricerca di un valore in un array `a` con `NUM_ELEM` elementi (risp. conteggio delle occorrenze):

```
for (i = trovato = 0;              /* for (i = contatore = 0;        */
     (i < NUM_ELEM && !trovato);   /*      (i < NUM_ELEM);           */
     i++)                          /*      i++)                      */
  trovato = a[i] == valore;        /*   contatore += a[i] == valore; */
```

- Un parametro formale di tipo array può essere dichiarato come segue:
  ◁*tipo elementi*▷ ◁*identificatore parametro*▷[]
  oppure nel seguente modo:
  `const` ◁*tipo elementi*▷ ◁*identificatore parametro*▷[]
  dove:
  - Ogni parametro effettivo di tipo array è passato per indirizzo, in quanto l'identificatore di una variabile di tipo array rappresenta l'indirizzo del primo elemento dell'array in forma simbolica. Ciò significa che le modifiche apportate ai valori contenuti in un parametro formale di tipo array durante l'esecuzione di una funzione vengono effettuate direttamente sui valori contenuti nel corrispondente parametro effettivo di tipo array.
  - Poiché non viene effettuata una copia di un parametro effettivo di tipo array, non è richiesta la specifica del numero di elementi del corrispondente parametro formale di tipo array (se serve, tale numero viene passato come ulteriore parametro).

  - Il qualificatore `const` stabilisce che i valori contenuti nel parametro formale di tipo array non possono essere modificati durante l'esecuzione della funzione. Ciò garantisce che i valori contenuti nel corrispondente parametro effettivo passato per indirizzo non vengano modificati durante l'esecuzione della funzione.

- Un array può avere più dimensioni:
  - La dichiarazione di una variabile di tipo array multidimensionale deve specificare il numero di elementi racchiuso tra parentesi quadre per ciascuna dimensione (p.e. `int tabella[2][3];`).
  - Nel caso di dichiarazione con contestuale inizializzazione, i valori debbono essere racchiusi entro parentesi graffe rispetto a ogni dimensione (p.e. `int tabella[2][3] = {{7, 9, 4}, {1, 5, 8}};`).
  - Il numero complessivo di elementi è il prodotto dei numeri di elementi per ciascuna dimensione.
  - La selezione di un elemento di una variabile di tipo array multidimensionale deve specificare l'indice racchiuso tra parentesi quadre per ciascuna dimensione (p.e. `tabella[i][j]`).
  - La dichiarazione di un parametro formale di tipo array multidimensionale deve specificare il numero di elementi per ciascuna dimensione tranne la prima e non può contenere `const`. ■fltpp_15

- Esempio di programma: statistica delle vendite.
  1. **Specifica del problema**. Calcolare il totale delle vendite effettuate da ogni agente in ogni stagione sulla base delle registrazioni delle singole vendite (agente, stagione, importo) contenute in un apposito file, come pure i totali per agente e per stagione rispettivamente.
  2. **Analisi del problema**. I dati di ingresso del problema sono rappresentati dalle registrazioni delle singole vendite contenute in un apposito file. I dati di uscita del problema sono rappresentati dal totale delle vendite effettuate da ogni agente in ogni stagione, più i totali per agente e per stagione. La relazione da sfruttare è che ciascun totale è dato dalla somma degli importi delle registrazioni relative all'agente e/o alla stagione in esame.
  3. **Progettazione dell'algoritmo**. Le registrazioni delle singole vendite contenute nel file – quindi accessibili solo in modo sequenziale – debbono essere preventivamente trasferite in una struttura dati che agevoli il calcolo dei totali per agente e per stagione, onde evitare una lettura completa dell'intero file per ogni coppia agente-stagione. A tale scopo, risulta particolarmente adeguata una struttura dati di tipo array bidimensionale – i cui elementi siano indicizzati da agenti e stagioni – che viene riempita man mano che si procede con la lettura delle registrazioni delle singole vendite dal file. Chiameremo questa struttura la tabella delle vendite.
     I passi dell'algoritmo – realizzabili attraverso altrettante funzioni – sono i seguenti:
     - Azzerare la tabella delle vendite e i totali per agente e per stagione.
     - Trasferire le registrazioni dal file delle vendite alla tabella delle vendite.
     - Calcolare i totali delle vendite per agente.
     - Calcolare i totali delle vendite per stagione.
     - Stampare la tabella delle vendite e i totali per agente e per stagione.
  4. **Implementazione dell'algoritmo**. Questa è la traduzione dei passi in C:

```
/*******************************************/
/* programma per la statistica delle vendite */
/*******************************************/

/*****************************/
/* inclusione delle librerie */
/*****************************/

#include <stdio.h>

/*****************************************/
/* definizione delle costanti simboliche */
/*****************************************/

#define AGENTI       9              /* numero di agenti che effettuano vendite */
#define STAGIONI     4              /* numero di stagioni */
#define FILE_VENDITE "vendite.txt"  /* nome fisico del file delle vendite */
```

```
/************************/
/* definizione dei tipi */
/************************/

typedef enum {autunno,
              inverno,
              primavera,
              estate}    stagione_t;   /* tipo stagione */

/*********************************/
/* dichiarazione delle funzioni */
/*********************************/

void azzera_strutture(double tabella_vendite[][STAGIONI],
                      double totali_agente[],
                      double totali_stagione[]);
int  trasf_reg_vendite(double tabella_vendite[][STAGIONI]);
void calc_tot_agente(double tabella_vendite[][STAGIONI],
                     double totali_agente[]);
void calc_tot_stagione(double tabella_vendite[][STAGIONI],
                       double totali_stagione[]);
void stampa_strutture(      double tabella_vendite[][STAGIONI],
                      const double totali_agente[],
                      const double totali_stagione[]);

/*******************************/
/* definizione delle funzioni */
/*******************************/

/* definizione della funzione main */
int main(void)
{
  /* dichiarazione delle variabili locali alla funzione */
  double tabella_vendite[AGENTI][STAGIONI],   /* output: tabella delle vendite */
         totali_agente[AGENTI],               /* output: totali per agente */
         totali_stagione[STAGIONI];           /* output: totali per stagione */
  int    trasferimento_riuscito;              /* lavoro: esito del trasferimento */

  /* azzerare la tabella delle vendite e i totali per agente e per stagione */
  azzera_strutture(tabella_vendite,
                   totali_agente,
                   totali_stagione);

  /* trasferire le registrazioni dal file delle vendite alla tabella delle vendite */
  trasferimento_riuscito = trasf_reg_vendite(tabella_vendite);

  if (trasferimento_riuscito)
  {
    /* calcolare i totali delle vendite per agente */
    calc_tot_agente(tabella_vendite,
                    totali_agente);

    /* calcolare i totali delle vendite per stagione */
    calc_tot_stagione(tabella_vendite,
                      totali_stagione);
```

```
    /* stampare la tabella delle vendite e i totali per agente e per stagione */
    stampa_strutture(tabella_vendite,
                     totali_agente,
                     totali_stagione);
  }
  return(0);
}

/* definizione della funzione per azzerare la tabella delle vendite
   e i totali per agente e per stagione */
void azzera_strutture(double tabella_vendite[][STAGIONI],   /* output: tab. vendite */
                      double totali_agente[],               /* output: tot. agente */
                      double totali_stagione[])             /* output: tot. stagione */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per agenti */
  stagione_t j;   /* lavoro: indice per stagioni */

  /* azzerare la tabella delle vendite */
  for (i = 0;
       (i < AGENTI);
       i++)
    for (j = autunno;
         (j <= estate);
         j++)
      tabella_vendite[i][j] = 0;

  /* azzerare i totali per agente */
  for (i = 0;
       (i < AGENTI);
       i++)
    totali_agente[i] = 0;

  /* azzerare i totali per stagione */
  for (j = autunno;
       (j <= estate);
       j++)
    totali_stagione[j] = 0;
}

/* definizione della funzione per trasferire le registrazioni dal file delle vendite
   alla tabella delle vendite */
int trasf_reg_vendite(double tabella_vendite[][STAGIONI])   /* output: tab. vendite */
{
  /* dichiarazione delle variabili locali alla funzione */
  FILE       *file_vendite;             /* input: file delle vendite */
  int         agente;                   /* input: agente letto nella registrazione */
  stagione_t  stagione;                 /* input: stagione letta nella registrazione */
  double      importo;                  /* input: importo letto nella registrazione */
  int         trasferimento_riuscito;   /* output: esito del trasferimento */

  /* aprire il file delle vendite */
  file_vendite = fopen(FILE_VENDITE,
                       "r");
```

```
  if (file_vendite == NULL)
  {
    trasferimento_riuscito = 0;
    printf("File delle vendite non trovato!\n");
  }
  else
  {
    trasferimento_riuscito = 1;
    /* trasferire le registrazioni dal file delle vendite alla tabella delle vendite */
    while (fscanf(file_vendite,
                  "%d%d%lf",
                  &agente,
                  (int *)&stagione,
                  &importo) != EOF)
      tabella_vendite[agente][stagione] += importo;
    /* chiudere il file delle vendite */
    fclose(file_vendite);
  }
  return(trasferimento_riuscito);
}
/* definizione della funzione per calcolare i totali delle vendite per agente */
void calc_tot_agente(double tabella_vendite[][STAGIONI],   /* input: tab. vendite */
                     double totali_agente[])               /* output: tot. agente */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per agenti */
  stagione_t j;   /* lavoro: indice per stagioni */
  /* calcolare i totali delle vendite per agente */
  for (i = 0;
       (i < AGENTI);
       i++)
    for (j = autunno;
         (j <= estate);
         j++)
      totali_agente[i] += tabella_vendite[i][j];
}
/* definizione della funzione per calcolare i totali delle vendite per stagione */
void calc_tot_stagione(double tabella_vendite[][STAGIONI],   /* input: tab. vendite */
                       double totali_stagione[])             /* output: tot. stagione */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per agenti */
  stagione_t j;   /* lavoro: indice per stagioni */
  /* calcolare i totali delle vendite per stagione */
  for (j = autunno;
       (j <= estate);
       j++)
    for (i = 0;
         (i < AGENTI);
         i++)
      totali_stagione[j] += tabella_vendite[i][j];
}
```

```
/* definizione della funzione per stampare la tabella delle vendite
   e i totali per agente e per stagione */
void stampa_strutture(      double tabella_vendite[][STAGIONI],   /* input: tab. vendite */
                      const double totali_agente[],               /* input: tot. agente */
                      const double totali_stagione[])             /* input: tot. stagione */
{
  /* dichiarazione delle variabili locali alla funzione */
  int        i;   /* lavoro: indice per agenti */
  stagione_t j;   /* lavoro: indice per stagioni */

  /* stampare l'intestazione di tutte le colonne */
  printf("Agente   Autunno   Inverno Primavera    Estate    Totale\n");

  /* stampare la tabella delle vendite e i totali per agente */
  for (i = 0;
       (i < AGENTI);
       i++)
  {
    printf("%6d  ",
           i);
    for (j = autunno;
         (j <= estate);
         j++)
      printf("%8.2f  ",
             tabella_vendite[i][j]);
    printf("%8.2f  \n",
           totali_agente[i]);
  }

  /* stampare l'intestazione dell'ultima riga */
  printf("Totale  ");

  /* stampare i totali per stagione */
  for (j = autunno;
       (j <= estate);
       j++)
    printf("%8.2f  ",
           totali_stagione[j]);
  printf("\n");
}
```

6.9 Stringhe: Rappresentazione e Funzioni di Libreria

- Ogni valore stringa presente in un programma C viene espresso come sequenza di caratteri racchiusa tra virgolette nel file sorgente e rappresentato con un array di elementi di tipo `char` in memoria principale a tempo d'esecuzione, quindi alle stringhe si applica quanto detto per gli array a una singola dimensione.
- Diversamente dai valori dei tipi visti finora, i quali occupano tutti la stessa quantità di memoria, i valori di tipo stringa occupano quantità di memoria diverse a seconda del numero di caratteri che li compongono (ciò accade anche per i valori del tipo standard `FILE`).

- Una variabile array di tipo stringa stabilisce il numero massimo di caratteri che possono essere contenuti. Poiché il valore di tipo stringa contenuto nella variabile in un certo momento può avere un numero di caratteri inferiore al massimo prestabilito, la fine di questo valore deve essere esplicitamente marcata con un carattere speciale, che è '`\0`' (analogia con la costante simbolica standard `EOF`).
- Se la dichiarazione di una variabile di tipo stringa contiene anche l'inizializzazione della variabile, invece di esprimere i singoli caratteri del valore iniziale tra parentesi graffe è possibile indicare l'intero valore iniziale tra virgolette. Il carattere '`\0`' non va indicato nel valore iniziale in quanto viene aggiunto automaticamente all'atto della memorizzazione del valore nella variabile.
- Quando il valore di una variabile di tipo stringa viene acquisito tramite `scanf` o sue varianti, il carattere '`\0`' viene automaticamente aggiunto all'atto della memorizzazione del valore nella variabile. L'identificatore di tale variabile non necessita di essere preceduto dall'operatore "`&`" quando compare nella `scanf` in quanto, essendo di tipo array, rappresenta già un indirizzo.
- Quando il valore di una variabile di tipo stringa viene comunicato tramite `printf` o sue varianti, vengono considerati tutti e soli i caratteri che precedono il carattere '`\0`' nel valore della variabile.
- Questioni da tenere presente quando si utilizza una variabile di tipo stringa:
  - Lo spazio di memoria riservato alla variabile deve essere sufficiente per contenere il valore di tipo stringa più il carattere '`\0`' al termine di ciascun utilizzo della variabile. In particolare, quando si acquisisce un valore di tipo stringa, è meglio usare `scanf` con segnaposto `%`◁*numero*▷`s` (così verranno considerati solo i primi ◁*numero*▷ caratteri digitati dall'utente) oppure acquisire un solo carattere alla volta tramite `getchar` all'interno di un'istruzione di ripetizione (alla cui fine bisogna però impostare esplicitamente '`\0`'), in modo da essere sicuri che lo spazio di memoria riservato alla variabile in cui il valore sarà memorizzato sia abbastanza grande.
  - Poiché tutte le funzioni di libreria standard per le stringhe fanno affidamento sulla presenza del carattere '`\0`' alla fine del valore di tipo stringa contenuto nella variabile, è fondamentale che tale carattere sia presente all'interno dello spazio di memoria riservato alla variabile al termine di ciascun utilizzo della variabile.
- Esempi:
  - Definizione di una costante simbolica di tipo stringa:
    ```
    #define FILE_VENDITE "vendite.txt"
    ```
  - Dichiarazione con contestuale inizializzazione di una variabile di tipo stringa (il valore iniziale occupa soltanto 9 dei 19 caratteri disponibili considerando il '`\0`' finale):
    ```
    char messaggio[20] = "benvenuto";
    ```
  - Acquisizione da tastiera di un valore di tipo stringa:
    ```
    scanf("%19s",
          messaggio);
    ```
  - Dichiarazione con contestuale inizializzazione di una variabile di tipo array di stringhe (il mese col nome più lungo ha 9 caratteri, perciò nella dichiarazione c'è `10` per tenere conto del '`\0`' finale):
    ```
    char mese[12][10] = {"gennaio",
                         "febbraio",
                         "marzo",
                         "aprile",
                         "maggio",
                         "giugno",
                         "luglio",
                         "agosto",
                         "settembre",
                         "ottobre",
                         "novembre",
                         "dicembre"};
    ```

- Principali funzioni messe a disposizione dal linguaggio C per il tipo stringa con indicazione dei relativi file di intestazione di libreria standard (il tipo standard `size_t` è assimilabile al tipo predefinito `unsigned int`, mentre il tipo `char *` è assimilabile al tipo stringa per quanto vedremo in Sez. 6.11):
  - `size_t strlen(const char *s)` `<string.h>`
    Restituisce la lunghezza di `s`, cioè il numero di caratteri attualmente in `s` escluso '\0'.
  - `char *strcpy(char *s1, const char *s2)` `<string.h>`
    Copia il contenuto di `s2` in `s1` (che deve avere spazio sufficiente).
  - `char *strncpy(char *s1, const char *s2, size_t n)` `<string.h>`
    Copia i primi `n` caratteri di `s2` in `s1` (che deve avere spazio sufficiente).
  - `char *strcat(char *s1, const char *s2)` `<string.h>`
    Concatena il contenuto di `s2` a quello di `s1` (che deve avere spazio sufficiente).
  - `char *strncat(char *s1, const char *s2, size_t n)` `<string.h>`
    Concatena i primi `n` caratteri di `s2` a quelli di `s1` (che deve avere spazio sufficiente).
  - `int strcmp(const char *s1, const char *s2)` `<string.h>`
    Confronta i contenuti di `s1` ed `s2` sulla base dell'ordinamento lessicografico restituendo:
    * `-1` se `s1 < s2`,
    * `0` se `s1 = s2`,
    * `1` se `s1 > s2`,

    dove `s1 < s2` se:
    * `s1` è più corta di `s2` e tutti i caratteri di `s1` coincidono con i corrispondenti caratteri di `s2`, oppure
    * i primi `n` caratteri di `s1` ed `s2` coincidono a due a due ed `s1[n] < s2[n]` rispetto alla codifica conforme allo standard ANSI usata per i caratteri.
  - `int strncmp(const char *s1, const char *s2, size_t n)` `<string.h>`
    Come la precedente, considerando solo i primi `n` caratteri di `s1` ed `s2`.
  - `int sprintf(char *s, const char *formato`$_p$`, ◁`*espressioni*`▷)` `<stdio.h>`
    Scrive su `s` il formato espanso coi valori delle espressioni (in particolare, permette di convertire numeri in stringhe).
  - `int sscanf(const char *s, const char *formato`$_s$`, ◁`*indirizzi variabili*`▷)` `<stdio.h>`
    Legge i valori contenuti all'inizio di `s` secondo il formato e li pone nelle variabili (in particolare, permette di estrarre numeri da stringhe).
  - `int atoi(const char *s)` `<stdlib.h>`
    Estrae da `s` la parte intera del numero che `s` contiene al suo inizio (altrimenti 0).
  - `double atof(const char *s)` `<stdlib.h>`
    Estrae da `s` il numero reale che `s` contiene al suo inizio (altrimenti 0). ■fltpp_16

6.10 Strutture e Unioni: Rappresentazione e Operatore Punto

- Il costruttore di tipo struttura del linguaggio C – noto più in generale come record – dà luogo a un valore aggregato formato da un numero finito di elementi memorizzati consecutivamente ma non necessariamente dello stesso tipo. Per questo motivo gli elementi non saranno selezionabili mediante indici come negli array, ma dovranno essere singolarmente dichiarati e identificati.
- Una variabile di tipo struttura viene dichiarata come segue:
  `struct {◁`*dichiarazione elementi*`▷} ◁`*identificatore variabile*`▷;`
  oppure con contestuale inizializzazione:
  `struct {◁`*dichiarazione elementi*`▷} ◁`*identificatore variabile*`▷ = {◁`*sequenza valori*`▷};`
  dove:
  - Ogni elemento (o campo) è dichiarato come segue:
    `◁`*tipo elemento*`▷ ◁`*identificatore elemento*`▷;`
  - Se presenti, i valori di inizializzazione sono separati da virgole e vengono ordinatamente assegnati da sinistra a destra ai corrispondenti elementi a patto che i rispettivi tipi siano compatibili.

- Diversamente dal tipo array, una variabile di tipo struttura può comparire in entrambi i lati di un assegnamento – quindi il risultato di una funzione può essere di tipo struttura – e può essere passata sia per valore che per indirizzo a una funzione.
- Ogni elemento di una variabile di tipo struttura può essere selezionato all'interno di un'istruzione tramite il suo identificatore. La sintassi è:
  ◁*identificatore variabile*▷.◁*identificatore elemento*▷
- L'operatore punto, che è unario e postfisso, ha la stessa precedenza dell'operatore di indicizzazione.
- Esempi:
  - Definizione di un tipo per i pianeti del sistema solare (il pianeta col nome più lungo ha 8 caratteri):

```
typedef struct
{
  char   nome[9];           /* nome del pianeta */
  double diametro;          /* diametro equatoriale in km */
  int    lune;              /* numero di lune */
  double tempo_orbita,      /* durata dell'orbita attorno al sole in anni */
         tempo_rotazione;   /* durata della rotazione attorno all'asse in ore */
} pianeta_t;
```

  - Dichiarazione con contestuale inizializzazione di una variabile per un pianeta:

```
pianeta_t pianeta = {"Giove",
                     142800,
                     16,
                     11.9,
                     9.925};
```

  - Acquisizione da tastiera dei dati relativi a un pianeta:

```
pianeta_t pianeta;

scanf("%8s%lf%d%lf%lf",
      pianeta.nome,
      &pianeta.diametro,
      &pianeta.lune,
      &pianeta.tempo_orbita,
      &pianeta.tempo_rotazione);
```

  - Funzione per verificare l'uguaglianza del contenuto di due variabili di tipo `pianeta_t`:

```
int pianeti_uguali(pianeta_t p1,   /* input: primo pianeta */
                   pianeta_t p2)   /* input: secondo pianeta */
{
  return(strcmp(p1.nome,
                p2.nome) == 0 &&
         p1.diametro == p2.diametro &&
         p1.lune == p2.lune &&
         p1.tempo_orbita == p2.tempo_orbita &&
         p1.tempo_rotazione == p2.tempo_rotazione);
}
```

  - Definizione di un tipo per i numeri complessi in forma algebrica (sebbene i due elementi siano entrambi numeri reali, essi hanno ruoli ben diversi e ciò giustifica il ricorso a una struttura piuttosto che a un array):

```
typedef struct
{
  double parte_reale,   /* parte reale del numero complesso */
         parte_immag;   /* parte immaginaria del numero complesso */
} num_compl_t;
```

  - Funzione per calcolare la somma di due numeri complessi in forma algebrica:

```
num_compl_t somma_num_compl(num_compl_t n1,   /* input: primo numero complesso */
                            num_compl_t n2)   /* input: secondo numero complesso */
{
  num_compl_t n;   /* output: somma dei due numeri complessi */

  n.parte_reale = n1.parte_reale + n2.parte_reale;
  n.parte_immag = n1.parte_immag + n2.parte_immag;
  return(n);
}
```

- Il costruttore di tipo unione del linguaggio C – noto più in generale come record variant – dà luogo a un valore aggregato formato da un numero finito di elementi non necessariamente dello stesso tipo, i quali sono in alternativa tra di loro. Ciò consente di rappresentare dati che possono avere interpretazioni diverse a seconda dei dati di ingresso del programma.
- Mentre lo spazio di memoria da riservare a una variabile di tipo struttura è la somma delle `sizeof` dei tipi degli elementi della struttura, lo spazio di memoria da riservare a una variabile di tipo unione è il massimo delle `sizeof` dei tipi degli elementi dell'unione, perché in questo caso i vari elementi sono in alternativa tra loro e quindi basta una quantità di spazio tale da contenere l'elemento più grande.
- La dichiarazione di una variabile di tipo unione ha la stessa sintassi della dichiarazione di una variabile di tipo struttura, con `struct` sostituito da `union`.
- Ogni elemento di una variabile di tipo unione può essere selezionato all'interno di un'istruzione tramite il suo identificatore utilizzando l'operatore punto. Per garantire che tale elemento sia coerente con l'interpretazione corrente della variabile, occorre testare preventivamente un'ulteriore variabile contenente l'interpretazione corrente.
- Esempi:
  - Definizione di un tipo per alcune figure geometriche piane (per chiarezza iniziamo dalla `typedef` che, per il compilatore, deve essere introdotta per ultima in quanto fa uso di tutte le altre; i valori degli elementi `forma` e `dati_figura` verranno acquisiti, mentre i valori degli elementi `perimetro` e `area` verranno calcolati sulla base dei precedenti):

```
typedef struct
{
  forma_t forma;         /* forma della figura (funge da interpretazione corrente) */
  union
  {
    triangolo_t  dati_triangolo;    /* dati del triangolo */
    rettangolo_t dati_rettangolo;   /* dati del rettangolo */
    double       lato_quadrato;     /* lunghezza del lato del quadrato */
    double       raggio_cerchio;    /* lunghezza del raggio del cerchio */
  }       dati_figura;   /* dati della figura */
  double  perimetro,     /* perimetro della figura */
          area;          /* area della figura */
} figura_t;
```

```
typedef enum {triangolo,
              rettangolo,
              quadrato,
              cerchio}    forma_t;

typedef struct
{
  double lato1,     /* lunghezza del primo lato */
         lato2,     /* lunghezza del secondo lato */
         lato3,     /* lunghezza del terzo lato */
         altezza;   /* altezza riferita al primo lato come base */
} triangolo_t;

typedef struct
{
  double lato1,   /* lunghezza del primo lato */
         lato2;   /* lunghezza del secondo lato */
} rettangolo_t;
```

– Funzione per calcolare il perimetro e l'area di una delle precedenti figure (che deve essere passata per indirizzo perché i risultati vengono memorizzati nei suoi elementi `perimetro` e `area`):

```
void calcola_perimetro_area(figura_t *fig)   /* input/output: figura */
{
  switch ((*fig).forma)
  {
    case triangolo:
      (*fig).perimetro = (*fig).dati_figura.dati_triangolo.lato1 +
                         (*fig).dati_figura.dati_triangolo.lato2 +
                         (*fig).dati_figura.dati_triangolo.lato3;
      (*fig).area = (*fig).dati_figura.dati_triangolo.lato1 *
                    (*fig).dati_figura.dati_triangolo.altezza / 2;
      break;
    case rettangolo:
      (*fig).perimetro = 2 * ((*fig).dati_figura.dati_rettangolo.lato1 +
                              (*fig).dati_figura.dati_rettangolo.lato2);
      (*fig).area = (*fig).dati_figura.dati_rettangolo.lato1 *
                    (*fig).dati_figura.dati_rettangolo.lato2;
      break;
    case quadrato:
      (*fig).perimetro = 4 * (*fig).dati_figura.lato_quadrato;
      (*fig).area = (*fig).dati_figura.lato_quadrato *
                    (*fig).dati_figura.lato_quadrato;
      break;
    case cerchio:
      (*fig).perimetro = 2 * PI_GRECO * (*fig).dati_figura.raggio_cerchio;
      (*fig).area = PI_GRECO * (*fig).dati_figura.raggio_cerchio *
                               (*fig).dati_figura.raggio_cerchio;
      break;
  }
}
```

Osserviamo che `*fig` deve essere racchiuso tra parentesi tonde perché altrimenti l'operatore punto verrebbe applicato prima dell'operatore valore-di "`*`" e quindi cercherebbe di selezionare un elemento da un parametro formale che non contiene un valore di tipo struttura, ma l'indirizzo di un valore di tipo struttura. Vedremo nella Sez. 6.11 un operatore che combina l'operatore punto e l'operatore valore-di, evitando il ricorso alle parentesi tonde in situazioni come questa. ■ftpp_17

6.11 Puntatori: Operatori e Funzioni di Libreria

- Il costruttore di tipo puntatore del linguaggio C viene utilizzato per denotare indirizzi di memoria, che sono espressi da numeri interi. Un valore di tipo puntatore non rappresenta quindi un dato, ma l'indirizzo di memoria al quale un dato può essere reperito.
- L'insieme dei valori di tipo puntatore include un valore speciale, denotato con la costante simbolica `NULL` definita nel file di intestazione di libreria standard `stdio.h`, il quale rappresenta l'assenza di un indirizzo specifico.
- Una variabile di tipo puntatore a un dato di un certo tipo viene dichiarata come segue:
  ◁*tipo dato*▷ *◁*identificatore variabile*▷;
  oppure con contestuale inizializzazione:
  ◁*tipo dato*▷ *◁*identificatore variabile*▷ = ◁*indirizzo*▷;
  Lo spazio di memoria da riservare a una variabile di tipo puntatore è indipendente dal tipo del dato a cui il puntatore fa riferimento.
- Poiché gli indirizzi di memoria vengono rappresentati attraverso numeri interi, i valori di tipo puntatore sono assimilabili ai valori di tipo `int`. Tuttavia, oltre all'operatore di assegnamento e al confronto di uguaglianza/diversità, ai valori di tipo puntatore è ragionevole applicare solo alcuni degli operatori aritmetico-logici:
  - Addizione/sottrazione di un valore di tipo `int` a/da un valore di tipo puntatore: serve per far avanzare/indietreggiare il puntatore di un certo numero di porzioni di memoria, ciascuna della dimensione tale da poter contenere un valore del tipo di riferimento del puntatore.
  - Sottrazione di un valore di tipo puntatore da un altro valore di tipo puntatore: serve per calcolare il numero di porzioni di memoria – ciascuna della dimensione tale da poter contenere un valore del tipo di riferimento dei due puntatori – comprese tra i due puntatori.
- Esistono inoltre degli operatori specifici per i valori di tipo puntatore:
  - L'operatore valore-di "`*`", applicato a un'espressione di tipo puntatore il cui valore è diverso da `NULL`, restituisce il valore contenuto nella locazione di memoria il cui indirizzo è dato dall'espressione. Se viene applicato a un'espressione di tipo puntatore il cui valore è `NULL`, il sistema operativo interrompe l'esecuzione del programma ed emette un messaggio d'errore.
  - L'operatore indirizzo-di "`&`", applicato a una variabile, restituisce l'indirizzo della locazione di memoria in cui è contenuto il valore della variabile (passaggio per indirizzo della variabile).
  - L'operatore freccia "`->`", il quale riguarda le variabili di tipo puntatore a struttura o unione, consente di abbreviare:
    (*◁*identificatore variabile puntatore*▷).◁*identificatore elemento*▷
    in:
    ◁*identificatore variabile puntatore*▷->◁*identificatore elemento*▷

  I primi due operatori sono unari e prefissi e hanno la stessa precedenza degli operatori unari aritmetico-logici (vedi Sez. 3.11), mentre l'operatore freccia è unario e postfisso e ha la stessa precedenza degli operatori di indicizzazione e punto (avremmo potuto usarlo nella funzione alla fine della Sez. 6.10).
- Poiché l'identificatore di una variabile di tipo array rappresenta l'indirizzo della locazione di memoria che contiene il valore del primo elemento dell'array, c'è un legame stretto tra array e puntatori:
  - Il valore di un elemento di una variabile di tipo array è normalmente selezionato come segue:
    ◁*identificatore variabile array*▷[◁*espr_indice*▷]
    Poiché il suo indirizzo è:
    ◁*identificatore variabile array*▷ + ◁*espr_indice*▷
    il suddetto valore può essere equivalentemente espresso come segue:
    *(◁*identificatore variabile array*▷ + ◁*espr_indice*▷)

  - Un parametro formale di tipo array può essere indifferentemente dichiarato come segue:
    ◁*tipo elementi*▷ ◁*identificatore parametro array*▷`[]`
    oppure nel seguente modo che è coerente col passaggio per indirizzo degli array:
    ◁*tipo elementi*▷ `*`◁*identificatore parametro array*▷
    Quest'ultimo è utilizzabile anche per dichiarare variabili array la cui lunghezza è ignota.

- Esempio: se una variabile stringa è dichiarata di tipo `char *` anziché array di `char`, allora il suo valore può essere acquisito tramite `scanf` usando il segnaposto `%ms`, che alloca anche lo spazio necessario, e anteponendo l'operatore "`&`" all'identificatore della variabile.

- Il costruttore di tipo struttura e il costruttore di tipo puntatore usati congiuntamente consentono la definizione di tipi di dati ricorsivi. Il costruttore di tipo struttura permette infatti di associare un identificatore alla struttura stessa mediante la seguente sintassi:
  `struct` ◁*identificatore struttura*▷ `{`◁*dichiarazione elementi*▷`}`
  In via di principio, ciò rende possibile la presenza di uno o più elementi della seguente forma all'interno di `{`◁*dichiarazione elementi*▷`}`:
  `struct` ◁*identificatore struttura*▷ ◁*identificatore elemento*▷`;`
  come pure di elementi della seguente forma:
  `struct` ◁*identificatore struttura*▷ `*`◁*identificatore elemento*▷`;`
  Tuttavia, solo gli elementi della seconda forma sono ammissibili in quanto rendono la definizione ricorsiva ben posta. Il motivo è che per gli elementi di questa forma è noto lo spazio di memoria da riservare, mentre ciò non vale per gli elementi della prima forma.

- Esempio di tipo di dato ricorsivo costituito dalla lista ordinata di numeri interi (o è vuota, o è composta da un numero intero collegato a una lista ordinata di numeri interi di valore maggiore del numero che li precede), i cui valori sono accessibili solo se si conosce l'indirizzo della sua prima componente:

```
typedef struct comp_lista
{
  int               valore;   /* numero intero memorizzato nella componente */
  struct comp_lista *succ_p;  /* puntatore alla componente successiva */
} comp_lista_t;
```

- Le strutture dati dinamiche, che si espandono e si contraggono mentre il programma viene eseguito, sono tipicamente implementate attraverso la definizione di tipi ricorsivi. Poiché lo spazio di memoria di cui queste strutture dati necessitano non può essere fissato a priori, l'allocazione/disallocazione della memoria per esse avviene a tempo di esecuzione nello heap (vedi Sez. 5.9) attraverso l'invocazione delle seguenti funzioni, basate su puntatori, disponibili nel file di intestazione di libreria standard `stdlib.h`:
  - `void *malloc(size_t dim)`
    Alloca un blocco di `dim` byte nello heap e restituisce l'indirizzo di tale blocco (`NULL` in caso di fallimento, cioè assenza di un blocco sufficientemente grande). Il blocco allocato viene marcato come occupato nello heap e il puntatore ad esso è del tipo generico `void *`.
  - `void *calloc(size_t num, size_t dim)`
    Alloca `num` blocchi consecutivi di `dim` byte ciascuno nello heap e restituisce l'indirizzo del primo blocco (`NULL` in caso di fallimento). Questa funzione serve per allocare dinamicamente un array nel momento in cui il numero dei suoi elementi diviene noto a tempo di esecuzione. Equivale a una `malloc` con parametro effettivo dato da `num * dim`.
  - `void *realloc(void *vecchio_blocco, size_t nuova_dim)`
    Cambia la dimensione di un blocco di memoria nello heap precedentemente allocato con `malloc`/`calloc` (senza modificarne il contenuto) e restituisce l'indirizzo del blocco ridimensionato (`NULL` in caso di fallimento). Quest'ultimo blocco potrebbe trovarsi in una posizione dello heap diversa da quella del blocco originario.
  - `void free(void *blocco)`
    Disalloca un blocco di memoria nello heap precedentemente allocato con `malloc`/`calloc`; prima di applicarla, è bene assicurarsi che non venga più fatto riferimento al blocco tramite puntatori nelle istruzioni da eseguire successivamente. Il blocco disallocato viene marcato come libero nello heap, ritornando quindi nuovamente disponibile per successive allocazioni.

- Esempi:
  - Allocazione dinamica e utilizzo di un array:

```
int *a,   /* array da allocare dinamicamente */
    n,    /* numero di elementi dell'array */
    i;    /* indice di scorrimento dell'array */

do
{
  printf("Digita il numero di elementi (> 0): ");
  scanf("%d",
        &n);
}
while (n <= 0);
a = (int *)calloc(n,              /* a = (int *)calloc(n + 1,        */
                  sizeof(int));   /*                   sizeof(int)); */
...                               /* a[0] = n;                       */
for (i = 0;                       /* for (i = 1;                     */
     (i < n);                     /*      (i <= n);                  */
     i++)                         /*      i++)                       */
  a[i] = 2 * i;                   /*   a[i] = 2 * i;                 */
```

    Sarebbe stato un errore dichiarare l'array dinamico nel seguente modo in quanto tutti gli operandi che compaiono nell'espressione che definisce il numero di elementi di un array debbono essere costanti (quindi una variabile come `n` non è ammissibile in quell'espressione):

```
int n,      /* numero di elementi dell'array */
    i,      /* indice di scorrimento dell'array */
    a[n];   /* array dinamico */
```

  - Funzione per attraversare una lista ordinata e stamparne i valori:

```
void attraversa_lista(comp_lista_t *testa_p)   /* input: indirizzo prima componente */
{
  comp_lista_t *punt;   /* lavoro: puntatore di scorrimento della lista */

  for (punt = testa_p;
       (punt != NULL);
       punt = punt->succ_p)
    printf("%d\n",
           punt->valore);
}
```

  - Funzione per cercare un valore in una lista ordinata (notare l'importanza della cortocircuitazione dell'operatore logico presente nelle istruzioni `for` e `if`):

```
comp_lista_t *cerca_in_lista(comp_lista_t *testa_p,   /* input: indirizzo prima comp. */
                             int           valore)    /* input: valore da cercare */
{
  comp_lista_t *punt;    /* output: puntatore di scorrimento della lista */

  for (punt = testa_p;
       (punt != NULL && punt->valore < valore);
       punt = punt->succ_p);
  if (punt != NULL && punt->valore > valore)
    punt = NULL;
  return(punt);
}
```

– Funzione per inserire un valore in una lista ordinata (l'indirizzo della prima componente potrebbe cambiare a seguito dell'inserimento, quindi deve essere passato per indirizzo da cui il doppio "*"):

```
int inserisci_in_lista(comp_lista_t **testa_p,   /* i/o: indirizzo prima componente */
                       int            valore)    /* input: valore da inserire */
{
  int           ris;       /* output: esito inserimento */
  comp_lista_t *corr_p,    /* lavoro: primo puntatore di scorrimento della lista */
               *prec_p,    /* lavoro: secondo puntatore di scorrimento della lista */
               *nuova_p;   /* lavoro: puntatore nuova componente */

  for (corr_p = prec_p = *testa_p;
       (corr_p != NULL && corr_p->valore < valore);
       prec_p = corr_p, corr_p = corr_p->succ_p);
  if (corr_p != NULL && corr_p->valore == valore)
    ris = 0;
  else
  {
    ris = 1;
    nuova_p = (comp_lista_t *)malloc(sizeof(comp_lista_t));
    nuova_p->valore = valore;
    nuova_p->succ_p = corr_p;
    if (corr_p == *testa_p)
      *testa_p = nuova_p;
    else
      prec_p->succ_p = nuova_p;
  }
  return(ris);
}
```

– Funzione per rimuovere un valore da una lista ordinata (l'indirizzo della prima componente potrebbe cambiare anche a seguito della rimozione, da cui di nuovo il doppio "*"):

```
int rimuovi_da_lista(comp_lista_t **testa_p,   /* i/o: indirizzo prima componente */
                     int            valore)    /* input: valore da rimuovere */
{
  int           ris;      /* output: esito rimozione */
  comp_lista_t *corr_p,   /* lavoro: primo puntatore di scorrimento della lista */
               *prec_p;   /* lavoro: secondo puntatore di scorrimento della lista */

  for (corr_p = prec_p = *testa_p;
       (corr_p != NULL && corr_p->valore < valore);
       prec_p = corr_p, corr_p = corr_p->succ_p);
  if (corr_p == NULL || corr_p->valore > valore)
    ris = 0;
  else
  {
    ris = 1;
    if (corr_p == *testa_p)
      *testa_p = corr_p->succ_p;
    else
      prec_p->succ_p = corr_p->succ_p;
    free(corr_p);
  }
  return(ris);
}
```

– Uso pericoloso di `free` (il problema si verifica anche con `calloc`/`realloc` al posto di `malloc`):

```
int *i,
    *j;

...
i = (int *)malloc(sizeof(int));
*i = 24;
j = i;
...
free(i);
...
*j = 18;
...
```

Le variabili `i` e `j` puntano alla stessa locazione di memoria, la quale contiene il valore `24` fino a quando non viene disallocata. Dopo la sua disallocazione, essa potrebbe essere nuovamente utilizzata per allocare qualche altra struttura dati, quindi assegnarle il valore `18` tramite la variabile di tipo puntatore `j` potrebbe causare l'effetto indesiderato di modificare accidentalmente il contenuto di un'altra variabile.

– Benché non contemplato dallo standard ANSI, il segnaposto `%ms` fu introdotto nel progetto GNU e poi standardizzato all'interno di POSIX proprio per semplificare l'acquisizione di un valore stringa da tastiera. Questo segnaposto consente di allocare esattamente lo spazio necessario a tempo d'esecuzione, evitando sia gli sprechi di memoria sia l'eccedenza della lunghezza massima prevista che spesso si manifestano nel caso di un array di `char` dichiarato staticamente, per il quale l'unico strumento a disposizione è l'utilizzo del segnaposto `%`◁*numero*▷`s` limitatamente al rispetto della lunghezza massima.

Il segnaposto `%ms` richiede che la variabile stringa sia dichiarata di tipo `char *` e che l'operatore "`&`" venga anteposto all'identificatore della variabile nella `scanf`:

```
char *stringa;

scanf("%ms",
      &stringa);
```

In assenza di `%ms`, per ottenere lo stesso effetto bisogna chiedere all'utente la lunghezza del valore stringa che vuole introdurre, allocare di conseguenza la variabile stringa e poi acquisire un carattere alla volta:

```
char *stringa;

stringa = (char *)calloc(lunghezza_stringa + 1,
                         sizeof(char));
for (i = 0;
     (i < lunghezza_stringa);
     i++)
{
  printf("Digita il carattere %d della stringa: ",
         i + 1);
  stringa[i] = getchar();
}
stringa[lunghezza_stringa] = '\0';
```

■fltpp_18

Capitolo 7

Correttezza di Programmi Procedurali

7.1 Stato della Computazione e Triple di Hoare

- Una volta terminato il suo sviluppo, il software non può essere subito rilasciato in quanto bisogna prima valutarne la correttezza. Questo può essere fatto ricorrendo ad attività di testing o di verifica:
  - Il *testing* richiede di individuare delle classi di dati di ingresso che siano rappresentative della maggior parte delle possibili esecuzioni del programma e osservare quali dati di uscita si ottengono per ciascuna di esse. Si tratta di un metodo sperimentale che non necessita di conoscenze matematiche, ma non esclude in generale la presenza di errori (maggiore è il numero di test effettuati, maggiore è il grado di copertura e quindi il livello di confidenza nella correttezza del programma).
  - La *verifica* richiede di sviluppare un modello matematico del programma, formalizzare le proprietà di interesse tramite formule logiche e stabilire se il modello soddisfa le formule. Si tratta di un metodo formale che necessita di conoscenze matematiche e garantisce l'assenza di errori specifici (più complesso è il programma, maggiore è il tempo necessario per verificarne la correttezza).
- Dati un problema e un programma, diciamo che *il programma è corretto rispetto al problema* se e solo se, *per ogni* istanza dei dati di ingresso del problema, l'esecuzione del programma *termina* e produce i dati di uscita corrispondenti all'istanza considerata.
- Ciò impone di determinare formalmente cosa il programma calcola. Ogni linguaggio di programmazione è dotato di un *dominio sintattico* (insieme dei programmi), un *dominio semantico* (insieme di oggetti matematici che formalizzano il significato dei programmi) e una *funzione di interpretazione* che associa a ogni programma il suo significato. Nel caso di un programma sequenziale, il suo significato viene tradizionalmente definito mediante una funzione matematica che descrive l'effetto ingresso/uscita dell'esecuzione del programma, ignorando tutti gli stati intermedi della computazione.
- Nel paradigma di programmazione imperativo di natura procedurale, per *stato della computazione* si intende il contenuto della memoria a un certo punto dell'esecuzione del programma. La funzione che rappresenta il significato del programma descrive quindi quale sia lo stato finale della computazione a fronte dello stato iniziale della computazione indotto da una generica istanza dei dati di ingresso.
- Tra i vari approcci alla verifica di correttezza dei programmi imperativi procedurali, l'approccio assiomatico di Hoare si basa sull'idea di annotare i programmi con formule della logica dei predicati (da ora in poi dette semplicemente predicati) che descrivono proprietà valide nei vari stati della computazione. Tali predicati esprimeranno asserzioni sui valori contenuti nelle variabili dei programmi.
- Si dice *tripla di Hoare* una tripla della seguente forma:
  $$\{Q\}\ S\ \{R\}$$
  dove Q è un predicato detto *precondizione*, S è un'istruzione ed R è un predicato detto *postcondizione*.
- La tripla $\{Q\}\ S\ \{R\}$ è vera se e solo se, iniziando in uno stato della computazione in cui Q è soddisfatta, l'esecuzione di S termina e raggiunge uno stato della computazione in cui R è soddisfatta.

7.2 Precondizione Più Debole e Regole di Dijkstra

- Nella pratica, data una tripla di Hoare $\{Q\}\ S\ \{R\}$ dove S è un intero programma, S è ovviamente noto come pure è nota la postcondizione R, la quale rappresenta in sostanza il risultato che si vuole ottenere alla fine dell'esecuzione del programma. La precondizione Q è invece ignota.
- Verificare la correttezza di un programma S che si prefigge di calcolare un risultato R consiste quindi nel determinare se esiste un predicato X che soddisfa la seguente equazione logica:
$$\{X\}\ S\ \{R\}\ \equiv\ \mathit{vero}$$
- Poiché l'equazione logica riportata sopra potrebbe ammettere più soluzioni, ci si concentra sulla determinazione della precondizione più debole (nel senso di meno vincolante), cioè la precondizione Q' tale che $Q'' \rightarrow Q'$ è vero per ogni precondizione Q'' che soddisfa l'equazione logica. Dati un programma S e una postcondizione R, denotiamo con $wp(S, R)$ la precondizione più debole rispetto a S ed R.
- Premesso che *vero* è soddisfatto da ogni stato della computazione mentre *falso* non è soddisfatto da nessuno stato della computazione, dati un programma S e una postcondizione R si hanno i seguenti tre casi:
  - Se $wp(S, R) = \mathit{vero}$, allora qualunque sia la precondizione Q risulta che la tripla di Hoare $\{Q\}\ S\ \{R\}$ è vera. Infatti, $Q \rightarrow \mathit{vero}$ è vero per ogni predicato Q. In tal caso, il programma è sempre corretto rispetto al problema, cioè è corretto a prescindere da quale sia lo stato iniziale della computazione.
  - Se $wp(S, R) = \mathit{falso}$, allora qualunque sia la precondizione Q risulta che la tripla di Hoare $\{Q\}\ S\ \{R\}$ è falsa. Infatti, $Q \rightarrow \mathit{falso}$ è vero se Q non è soddisfatto nello stato iniziale della computazione, mentre $Q \rightarrow \mathit{falso}$ è falso (e quindi Q non può essere una soluzione) se Q è soddisfatto nello stato iniziale della computazione. In tal caso, il programma non è mai corretto rispetto al problema, cioè non è corretto a prescindere da quale sia lo stato iniziale della computazione.
  - Se $wp(S, R) \notin \{\mathit{vero}, \mathit{falso}\}$, allora la correttezza del programma rispetto al problema potrebbe dipendere dallo stato iniziale della computazione.
- Le seguenti proprietà risultano utili durante la verifica di correttezza ($\equiv$ sta per $\Leftrightarrow$, $\models$ sta per $\Rightarrow$):
$$\begin{aligned} wp(S, \mathit{falso}) &\equiv \mathit{falso} \\ wp(S, R_1 \wedge R_2) &\equiv (wp(S, R_1) \wedge wp(S, R_2)) \\ (R_1 \rightarrow R_2) &\models (wp(S, R_1) \rightarrow wp(S, R_2)) \\ (\{Q\}\ S\ \{R'\} \wedge (R' \rightarrow R)) &\models \{Q\}\ S\ \{R\} \end{aligned}$$
- Dato un programma S privo di iterazione e ricorsione e data una postcondizione R, la terminazione è fuori discussione e $wp(S, R)$ può essere determinata per induzione sulla struttura sintattica di S mediante le seguenti *regole di Dijkstra*:
  1. Se S è un'istruzione di assegnamento "`x = e;`", si applica la seguente regola di retropropagazione:
  $$wp(S, R)\ =\ R_{\mathtt{x,e}}$$
  dove $R_{\mathtt{x,e}}$ è il predicato ottenuto da R sostituendo tutte le occorrenze di `x` con `e`.
  2. Se S è un'istruzione di selezione "`if` (β) S_1 `else` S_2", si ragiona sulle due parti come segue:
  $$wp(S, R)\ =\ ((\beta \rightarrow wp(S_1, R)) \wedge (\neg\beta \rightarrow wp(S_2, R)))$$
  3. Se S è una sequenza di istruzioni "$S_1 S_2$", si procede a ritroso come segue:
  $$wp(S, R)\ =\ wp(S_1, wp(S_2, R))$$

  Dalla seconda formula segue $((\beta \wedge wp(S_1, R)) \veebar (\neg\beta \wedge wp(S_2, R)))$ dopo aver sostituito le implicazioni.
- Come suggerito dalla regola per la sequenza di istruzioni, il calcolo della precondizione più debole per un intero programma procede andando a ritroso a partire dalla postcondizione e dall'ultima istruzione (la loro precondizione più debole diventa la postcondizione per la penultima istruzione e così via). L'applicazione delle regole suddette è superflua se la postcondizione ripete banalmente le istruzioni oggetto di verifica.

- Esempi:

  – La correttezza di un programma può dipendere dallo stato iniziale della computazione, nel qual caso la precondizione più debole non è equivalente né a *vero* né a *falso*. Data l'istruzione di assegnamento S:

  ```
  x = x + 1;
  ```

  e data la postcondizione R:

  $$x < 5$$

  l'ottenimento del risultato prefissato dipende ovviamente dal valore di `x` prima che venga eseguita l'istruzione. Infatti la precondizione più debole risulta essere:

  $$wp(S, R) \;=\; (x < 5)_{\mathtt{x},\mathtt{x+1}} \;=\; (x + 1 < 5) \;\equiv\; (x < 4)$$

  In conclusione, abbiamo formalmente dimostrato che l'esecuzione dell'istruzione considerata raggiunge uno stato della computazione in cui `x` contiene un valore minore di 5 solo partendo da uno stato della computazione in cui `x` contiene un valore minore di 4, come ci si aspettava.

  – Non c'è dipendenza dallo stato iniziale della computazione se la precondizione più debole è *vero*. Il seguente brano di codice S per determinare quale tra le variabili `x` e `y` contiene il valore minimo:

  ```
  if (x <= y)
    z = x;
  else
    z = y;
  ```

  è sempre corretto perché, formalizzando la postcondizione R nel seguente modo:

  $$z = \min(x, y)$$

  e indicando con β l'espressione "$x \leq y$", con S_1 l'istruzione "`z = x;`" e con S_2 l'istruzione "`z = y;`", la precondizione più debole risulta essere *vero* in quanto:

  $(\beta \rightarrow wp(S_1, R)) \;=\; (x \leq y \;\rightarrow\; (z = \min(x, y))_{\mathtt{z},\mathtt{x}}) \;=\; (x \leq y \;\rightarrow\; x = \min(x, y)) \;\equiv\; \mathit{vero}$
  $(\neg\beta \rightarrow wp(S_2, R)) \;=\; (x > y \;\rightarrow\; (z = \min(x, y))_{\mathtt{z},\mathtt{y}}) \;=\; (x > y \;\rightarrow\; y = \min(x, y)) \;\equiv\; \mathit{vero}$
  $wp(S, R) \;=\; ((\beta \rightarrow wp(S_1, R)) \wedge (\neg\beta \rightarrow wp(S_2, R))) \;\equiv\; (\mathit{vero} \wedge \mathit{vero}) \;\equiv\; \mathit{vero}$

  Una postcondizione come $(x \leq y \;\rightarrow\; z = x) \wedge (x > y \;\rightarrow\; z = y)$ sarebbe stata banale in quanto ripete il codice di S, mentre una postcondizione come $(z = x) \vee (z = y)$ sarebbe stata vaga perché non precisa quando `z` contiene quei valori.

  – Potrebbe non esserci dipendenza dallo stato iniziale della computazione anche se la precondizione più debole non è *vero*. Il seguente brano di codice S per scambiare i valori delle variabili `x` e `y`:

  ```
  tmp = x;
  x = y;
  y = tmp;
  ```

  è sempre corretto perché, indicando con X e Y i valori rispettivamente contenuti nelle variabili `x` e `y` all'inizio dell'esecuzione e formalizzando la postcondizione R nel seguente modo:

  $$x = Y \wedge y = X$$

  la precondizione più debole risulta essere proprio il predicato:

  $$x = X \wedge y = Y$$

  Infatti, indicando rispettivamente con S_1, S_2, S_3 le tre istruzioni di assegnamento che compongono S dall'alto verso il basso, si ha:

  $wp(S_3, R) \;=\; (x = Y \wedge y = X)_{\mathtt{y},\mathtt{tmp}} \;=\; (x = Y \wedge tmp = X)$ $[R']$
  $wp(S_2, R') \;=\; (x = Y \wedge tmp = X)_{\mathtt{x},\mathtt{y}} \;=\; (y = Y \wedge tmp = X)$ $[R'']$
  $wp(S_1, R'') \;=\; (y = Y \wedge tmp = X)_{\mathtt{tmp},\mathtt{x}} \;=\; (y = Y \wedge x = Y) \;\equiv\; (x = X \wedge y = Y)$

  In conclusione, abbiamo formalmente dimostrato che, partendo da uno stato della computazione in cui `x` contiene il valore X e `y` contiene il valore Y, l'esecuzione dell'istruzione S raggiunge uno stato della computazione in cui `x` contiene il valore Y e `y` contiene il valore X. Poiché questo vale a prescindere dagli specifici valori X e Y, l'istruzione S è sempre corretta. ■fltpp_19

7.3 Verifica della Correttezza di Programmi Procedurali Iterativi

- Per verificare mediante triple di Hoare la correttezza di un'istruzione di ripetizione, non è possibile procedere applicando meccanicamente le regole di Dijkstra in quanto bisogna preoccuparsi anche della terminazione dell'esecuzione dell'istruzione stessa (ricordiamo che Turing dimostrò che il problema della terminazione non è decidibile).

- La tecnica da utilizzare in questo caso consiste nell'individuare quanto segue basandosi sulle variabili di controllo del ciclo:

  – Un invariante di ciclo, cioè un predicato che è soddisfatto sia nello stato iniziale della computazione che nello stato finale della computazione di ciascuna iterazione.

  – Una funzione descrescente, che misura il tempo residuo alla fine dell'esecuzione dell'istruzione di ripetizione.

  Sotto certe ipotesi denominate invarianza, progresso e limitatezza, l'invariante di ciclo risulta essere una precondizione dell'istruzione di ripetizione, anche se non necessariamente la più debole.

- *Teorema dell'invariante di ciclo*: Data un'istruzione di ripetizione "`while (`β`)` S", se esistono un predicato P e una funzione intera tr tali che:

$$\begin{array}{ll} \{P \wedge \beta\}\ S\ \{P\} & \text{(invarianza)} \\ \{P \wedge \beta \wedge tr(i) = t\}\ S\ \{tr(i+1) < t\} & \text{(progresso)} \\ (P \wedge tr(i) \leq 0) \models \neg\beta & \text{(limitatezza)} \end{array}$$

  allora:

$$\{P\}\ \texttt{while (}\beta\texttt{)}\ S\ \{P \wedge \neg\beta\}$$

- Corollario: Date un'istruzione di ripetizione "`while (`β`)` S" e una postcondizione R, se esiste un invariante di ciclo P per quell'istruzione tale che:

$$(P \wedge \neg\beta) \models R$$

  allora:

$$\{P\}\ \texttt{while (}\beta\texttt{)}\ S\ \{R\}$$

- Esempio: il seguente programma per calcolare la somma dei numeri contenuti nei primi 10 elementi di un array:

```
somma = 0;
i = 0;
while (i <= 9)
{
  somma = somma + a[i];
  i = i + 1;
}
```

  è corretto perché, formalizzando la postcondizione nel seguente modo:

$$R\ =\ (somma = \sum_{j=0}^{9} a[j])$$

  si può rendere la tripla vera mettendo come precondizione *vero* in quanto:

  – Il predicato:

$$P\ =\ (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j])$$

    e la funzione:

$$tr(i)\ =\ 10 - i$$

    soddisfano le ipotesi del teorema dell'invariante di ciclo per l'istruzione `while` perché:

∗ L'invarianza:

$$\{P \wedge i \leq 9\}\ \texttt{somma = somma + a[i]; i = i + 1;}\ \{P\}$$

segue da:

$$\begin{aligned} P_{\texttt{i,i+1}} &= (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i+1-1} a[j]) \\ &\equiv (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i} a[j]) \end{aligned}$$

e, denotato con P' quest'ultimo predicato, da:

$$\begin{aligned} P'_{\texttt{somma,somma+a[i]}} &= (0 \leq i+1 \leq 10 \wedge somma + a[i] = \sum_{j=0}^{i} a[j]) \\ &\equiv (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j]) \end{aligned}$$

giacché, denotato con P'' quest'ultimo predicato, $(P \wedge i \leq 9)$ è una precondizione più forte:

$$\begin{aligned} (P \wedge i \leq 9) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge i \leq 9) \\ &\models P'' \end{aligned}$$

∗ Il progresso è garantito dal fatto che $tr(i)$ decresce di un'unità a ogni iterazione in quanto `i` viene incrementata di un'unità a ogni iterazione.

∗ La limitatezza segue da:

$$\begin{aligned} (P \wedge tr(i) \leq 0) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge 10 - i \leq 0) \\ &\equiv (i = 10 \wedge somma = \sum_{j=0}^{9} a[j]) \\ &\models (i \not\leq 9) \end{aligned}$$

– Poiché:

$$\begin{aligned} (P \wedge i \not\leq 9) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge i \not\leq 9) \\ &\equiv (i = 10 \wedge somma = \sum_{j=0}^{9} a[j]) \\ &\models R \end{aligned}$$

dal corollario del teorema dell'invariante di ciclo segue che P può essere usato come precondizione dell'intera istruzione `while` con la postcondizione originaria R.

– Proseguendo infine a ritroso rispetto alle due istruzioni che precedono il ciclo, si ottiene prima:

$$P_{\texttt{i,0}} = (0 \leq 0 \leq 10 \wedge somma = \sum_{j=0}^{0-1} a[j]) \equiv (somma = 0)$$

e poi, denotato con P''' quest'ultimo predicato, si ha:

$$P'''_{\texttt{somma,0}} = (0 = 0) \equiv vero$$

7.4 Verifica della Correttezza di Programmi Procedurali Ricorsivi

- Come nel caso dei programmi procedurali iterativi, così anche nel caso dei programmi procedurali ricorsivi non ci sono regole come quelle di Dijkstra da applicare meccanicamente. Inoltre, non è nemmeno detto che sia necessario utilizzare le triple di Hoare.
- Per verificare la correttezza di un programma ricorsivo, conviene ricorrere al principio di induzione, avvalendosi eventualmente anche delle triple di Hoare e delle regole di Dijkstra laddove appropriato.

- Nell'applicazione del principio di induzione alla verifica di correttezza di un programma ricorsivo, dopo aver formalizzato tramite una formula logica cosa il programma deve calcolare, bisogna dimostrare che questa formula è soddisfatta procedendo per induzione su una qualche grandezza contenuta nella formula stessa. Ciò significa individuare uno o più casi base e uno o più casi generali, a ciascuno dei quali corrisponderanno rami diversi del codice. A ogni ramo si applica la formula come postcondizione e poi si procede a ritroso con le regole di Dijkstra quando appropriato, sfruttando l'ipotesi induttiva nei casi generali.
- Esempi relativi ad alcune delle funzioni ricorsive della Sez. 5.8:
  - La funzione ricorsiva per calcolare il fattoriale di n:

```
int fattoriale(int n)   /* input: n >= 0 */
{
  int fatt;   /* output: risultato */

  if (n == 0)
    fatt = 1;
  else
    fatt = n * fattoriale(n - 1);
  return(fatt);
}
```

    soddisfa `fattoriale(n)` $= n!$ per ogni $n \in \mathbb{N}$ come si dimostra procedendo per induzione su n:
    - Sia $n = 0$. Risulta `fattoriale(0)` $= 1 = 0!$ e quindi l'asserto è vero per $n = 0$.
    - Dato un certo $n \geq 1$, supponiamo che `fattoriale(n - 1)` $= (n-1)!$. Risulta `fattoriale(n)` $=$ `n * fattoriale(n - 1)` $= n \cdot (n-1)!$ per ipotesi induttiva. Poiché $n \cdot (n-1)! = n!$, l'asserto è vero per n.
  - La funzione ricorsiva per calcolare l'n-esimo numero di Fibonacci:

```
int fibonacci(int n)   /* input: n >= 1 */
{
  int fib;   /* output: risultato */

  if (n == 1 || n == 2)
    fib = 1;
  else
    fib = fibonacci(n - 1) + fibonacci(n - 2);
  return(fib);
}
```

    soddisfa `fibonacci(n)` $= fib_n$ per ogni $n \geq 1$ come si dimostra procedendo per induzione su n:
    - Sia $n \in \{1, 2\}$. Risulta `fibonacci(n)` $= 1 = fib_n$ e quindi l'asserto è vero per $n \in \{1, 2\}$.
    - Dato un certo $n \geq 3$, supponiamo che `fibonacci(m)` $= fib_m$ per ogni m tale che $1 \leq m < n$. Risulta `fibonacci(n)` $=$ `fibonacci(n - 1)` $+$ `fibonacci(n - 2)` $= fib_{n-1} + fib_{n-2}$ per ipotesi induttiva. Poiché $fib_{n-1} + fib_{n-2} = fib_n$, l'asserto è vero per n.
  - La funzione ricorsiva per risolvere il problema delle torri di Hanoi rispetta le due regole date – (i) è possibile spostare un solo disco alla volta e (ii) un disco non può mai essere appoggiato sopra un disco di diametro inferiore – come si dimostra procedendo per induzione sugli $n \geq 1$ dischi:
    - Sia $n = 1$. Risulta che `hanoi(n, 'A', 'C', 'B')` soddisfa banalmente sia (i) che (ii) e quindi l'asserto è vero per $n = 1$.
    - Dato un certo $n \geq 2$, supponiamo che `hanoi(n - 1, p, a, i)` soddisfi (i) e (ii) per ogni tripla `(p, a, i)` di valori tratti da {`'A'`, `'B'`, `'C'`}. Risulta che `hanoi(n, p, a, i)` soddisfa (i) e (ii) in virtù dell'ipotesi induttiva applicata ad `hanoi(n - 1, p, i, a)` e `hanoi(n - 1, i, a, p)` e del fatto che la mossa tra le due chiamate ricorsive sposta il disco più grande verso l'asta `a` che in quel momento è vuota. Dunque l'asserto è vero per n.

- Esempio: il massimo e il submassimo di un insieme I_n contenente $n \geq 2$ numeri può essere determinato attraverso una funzione ricorsiva che a ogni invocazione dimezza l'insieme da esaminare e poi ne calcola massimo e submassimo confrontando massimi e submassimi delle sue due metà $I'_{n/2}$ e $I''_{n/2}$:

```
coppia_t calcola_max_submax(int a[],   /* input: array che implementa l'insieme */
                            int sx,    /* input: indice estremo sinistro (0 all'inizio) */
                            int dx)    /* input: indice estremo destro (n - 1 all'inizio) */
{
  coppia_t ms,    /* output: max e submax su a[sx], ..., a[dx] */
           ms1,   /* lavoro: max e submax su a[sx], ..., a[(sx + dx) / 2] */
           ms2;   /* lavoro: max e submax su a[(sx + dx) / 2 + 1], ..., a[dx] */

  if (dx - sx + 1 == 2)
  {
    ...   /* vedi sotto, caso n = 2 */
  }
  else
  {
    ms1 = calcola_max_submax(a,
                             sx,
                             (sx + dx) / 2);
    ms2 = calcola_max_submax(a,
                             (sx + dx) / 2 + 1,
                             dx);
    ...   /* vedi sotto, caso n > 2 */
  }
  return(ms);
}
```

Dimostriamo che `calcola_max_submax`(I_n) = max_submax(I_n), dove max_submax è la funzione matematica definita ponendo max_submax(I_n) = (max(I_n), max($I_n \setminus \{\max(I_n)\}$)), procedendo per induzione su n:

- Sia $n = 2$. Risulta `calcola_max_submax`(I_2) = max_submax(I_2) e quindi l'asserto è vero per $n = 2$, perché usando le triple di Hoare e andando a ritroso con le regole di Dijkstra si ha:

```
/* {vero} */
if (a[sx] >= a[dx])
{
  /* {a[sx] = max(I_2) /\ a[dx] = submax(I_2)} */
  ms.max = a[sx];
  /* {ms.max = max(I_2) /\ a[dx] = submax(I_2)} */
  ms.submax = a[dx];
}
else
{
  /* {a[dx] = max(I_2) /\ a[sx] = submax(I_2)} */
  ms.max = a[dx];
  /* {ms.max = max(I_2) /\ a[sx] = submax(I_2)} */
  ms.submax = a[sx];
}
/* {ms.max = max(I_2) /\ ms.submax = submax(I_2)} */
```

– Dato un certo $n > 2$, sia vero l'asserto per ogni m tale che $2 \leq m < n$. In questo caso, la funzione invoca ricorsivamente `calcola_max_submax(`$I'_{n/2}$`)` e `calcola_max_submax(`$I''_{n/2}$`)`, quindi per ipotesi induttiva `ms1` = max_submax($I'_{n/2}$) ed `ms2` = max_submax($I''_{n/2}$), rispettivamente. L'asserto è allora vero per n, perché usando le triple di Hoare e andando a ritroso con le regole di Dijkstra si ha:

```
/* {vero} */
if (ms1.max >= ms2.max)
{
  /* {ms1.max = max(I_n) /\
      (ms2.max >= ms1.submax --> ms2.max = submax(I_n)) /\
      (ms2.max < ms1.submax --> ms1.submax = submax(I_n))} */
  ms.max = ms1.max;
  if (ms2.max >= ms1.submax)
    /* {ms.max = max(I_n) /\ ms2.max = submax(I_n)} */
    ms.submax = ms2.max;
  else
    /* {ms.max = max(I_n) /\ ms1.submax = submax(I_n)} */
    ms.submax = ms1.submax;
}
else
{
  /* {ms2.max = max(I_n) /\
      (ms1.max >= ms2.submax --> ms1.max = submax(I_n)) /\
      (ms1.max < ms2.submax --> ms2.submax = submax(I_n))} */
  ms.max = ms2.max;
  if (ms1.max >= ms2.submax)
    /* {ms.max = max(I_n) /\ ms1.max = submax(I_n)} */
    ms.submax = ms1.max;
  else
    /* {ms.max = max(I_n) /\ ms2.submax = submax(I_n)} */
    ms.submax = ms2.submax;
}
/* {ms.max = max(I_n) /\ ms.submax = submax(I_n)} */
```

■fltpp_20

Capitolo 8

Attività di Laboratorio in Linux

8.1 Cenni di Storia di Linux

- Il sistema operativo Linux si ispira a Unix, il sistema operativo sviluppato a partire dai primi anni 1970 presso i Bell Lab della AT&T e successivamente diffusosi in ambito universitario. Nel 1983 Richard Stallman lanciò un progetto per creare GNU, un clone di Unix che fosse liberamente distribuibile e modificabile, così da sottrarre Unix ad aziende come Sun, SCO e IBM che lo stavano commercializzando. Nel 1991 il lavoro fu completato con l'implementazione del nucleo da parte di Linus Torvalds, da cui deriva il nome Linux. Dal punto di vista tecnologico, la principale novità introdotta da Linux è la distribuzione del software a livello di sorgenti organizzati in pacchetti.
- Linux, nelle sue varie distribuzioni (come ad esempio Debian, Ubuntu, Mint, Suse e RedHat), può essere utilizzato su qualsiasi personal computer, sia di tipo desktop che di tipo laptop, tanto che diversi costruttori hanno ormai messo sul mercato computer con Linux preinstallato al posto di sistemi operativi proprietari come Windows e MacOS. Grazie alla sua affidabilità, Linux è molto spesso impiegato su macchine server. Nel 2003 è stata sviluppata una versione modificata, nota come Android e acquisita da Google nel 2005, che è adottata dal 2008 da molti produttori di smartphone come sistema operativo concorrente di IOS.
- Linux è l'esempio più importante di software libero e open source. Altri esempi di questo genere sono il browser web Mozilla Firefox, il sistema web WordPress, gli strumenti di produttività per ufficio OpenOffice e LibreOffice, gli editor di testo Gvim ed Emacs, il client di posta elettronica Alpine, il sistema per l'apprendimento a distanza Moodle e il sistema di web conferencing BigBlueButton.
- Il software libero è soggetto a una licenza d'uso ma, diversamente dal software proprietario, garantisce quattro libertà fondamentali definite negli anni 1980 da Richard Stallman, il fondatore della Free Software Foundation. Esse sono:
  - la libertà di eseguire il programma per qualsiasi scopo;
  - la libertà di studiare il programma e di modificarlo;
  - la libertà di distribuire copie del programma a chiunque ne abbia bisogno;
  - la libertà di migliorare il programma e di distribuirne pubblicamente i miglioramenti in modo tale che tutti ne possano beneficiare.
- È rilevante osservare come essere a sorgente aperto (cioè ispezionabile), che è un aspetto tecnico, sia un prerequisito per essere un software libero, che è un aspetto etico e sociale che favorisce la nascita di comunità mondiali di sviluppatori che curano prodotti software per tutti. Va inoltre precisato che un software di questo genere non è necessariamente gratuito (freeware), anche se quasi sempre lo è.
- Comandi per ottenere informazioni su Linux o uno dei suoi comandi:

  ```
  uname -a
  man ◁comando▷
  ```

8.2 Gestione dei File e Principali Comandi in Linux

- In Linux i file sono organizzati in directory secondo una struttura gerarchica ad albero, in cui la radice è denotata con "`/`", i nodi interni corrispondono alle directory e le foglie corrispondono ai file. Ogni file è conseguentemente individuato dal suo nome di percorso, il quale è ottenuto facendo precedere il nome del file dalla concatenazione dei nomi delle directory che si incontrano lungo l'unico percorso nell'albero che va dalla radice al file. Tutti questi nomi sono separati da "`/`" nel nome di percorso.
- Ogni directory ha un nome di percorso formato allo stesso modo. Tuttavia, sono disponibili le seguenti tre abbreviazioni per i nomi di percorso delle directory:
  - "`.`" denota la directory di lavoro, cioè la directory dove l'utente sta lavorando.
  - "`..`" denota la directory genitrice della directory di lavoro.
  - "`~`" denota la home directory dell'utente.
- Il nome di un file o di una directory è una sequenza di lettere, cifre decimali, sottotratti e trattini ed è consigliabile che non contenga spazi al suo interno. Le lettere minuscole sono considerate diverse dalle corrispondenti lettere maiuscole (case sensitivity). Di solito il nome di un file contiene anche un'estensione che individua il tipo del file, come ad esempio:
  - `.c` per un file sorgente del linguaggio C.
  - `.h` per un file di intestazione di una libreria del linguaggio C.
  - `.o` per un file oggetto di un linguaggio compilato come il C.
  - `.html` per un file sorgente del linguaggio interpretato HTML.
  - `.txt` per un file di testo senza formattazione.
  - `.doc` per un file contenente un documento in formato Word.
  - `.ps` per un file contenente un documento in formato PostScript.
  - `.pdf` per un file contenente un documento in formato PDF.
  - `.tar` per un file risultante dall'accorpamento di più file in uno solo.
  - `.gz` per un file risultante dalla compressione del contenuto di un file.
  - `.zip` per un file risultante dalla compressione del contenuto di uno o più file.
- Poiché i comandi di Linux corrispondono a file eseguibili, non si danno estensioni ai file eseguibili cosicché nemmeno i comandi hanno delle estensioni. Inoltre, se un file è nascosto (per esempio un file di configurazione), il suo nome inizia con "`.`" e non va confuso con un'estensione.
- Esempio di nome di percorso di un file sorgente C:
  `/home/users/bernardo/programmi/conversione_mi_km.c`
- Ogni file ha ad esso associato alcune informazioni tra cui le seguenti:
  - Identificativo dell'utente proprietario del file (di solito è l'utente che ha creato il file).
  - Identificativo del gruppo di utenti proprietario del file (di solito è uno dei gruppi di cui l'utente che ha creato il file fa parte).
  - Dimensione del file espressa in Kbyte.
  - Data e ora in cui è avvenuta l'ultima modifica del file.
  - Diritti di accesso. Questi sono espressi attraverso tre triplette binarie che rappresentano i diritti di accesso dell'utente proprietario, del gruppo di utenti proprietario e del resto degli utenti, rispettivamente. In ciascuna tripletta il primo bit esprime il permesso ("`r`") o il divieto ("`-`") di lettura, il secondo bit il permesso ("`w`") o il divieto ("`-`") di scrittura e il terzo bit il permesso ("`x`") o il divieto ("`-`") di esecuzione.

- Le medesime informazioni sono associate a ogni directory, con le seguenti differenze:
  - La dimensione della directory è il numero di Kbyte necessari per memorizzare le informazioni che descrivono il contenuto della directory (quindi non è la somma delle dimensioni dei suoi file).
  - Il diritto di esecuzione va inteso per la directory come diritto di accesso alla directory (quindi determina la possibilità per un utente di avere quella directory come directory di lavoro).
- Comandi relativi alle directory:
  - Visualizza il nome di percorso della directory in cui l'utente sta lavorando:
    `pwd`
    All'inizio della sessione di lavoro, la directory di lavoro coincide con la home directory dell'utente.
  - Visualizza il contenuto di una directory:
    `ls` ◁*directory*▷ (elenco dei nomi di file e sottodirectory)
    `ls -lF` ◁*directory*▷ (elenco completo di tutte le informazioni)
    `ls -alF` ◁*directory*▷ (elenco completo comprensivo dei file nascosti)
    dove *directory* può essere omessa se è la directory di lavoro. Se il contenuto della directory non sta tutto in una schermata, aggiungere il seguente filtro al precedente comando:
    `| more`
    al fine di visualizzare una schermata per volta.
  - Visualizza l'occupazione su disco di una directory:
    `du -h` ◁*directory*▷
  - Cambia la directory di lavoro:
    `cd` ◁*directory*▷
    dove *directory* può essere omessa se è la home directory.
  - Crea una directory:
    `mkdir` ◁*directory*▷
  - Copia una directory in un'altra directory:
    `cp -r` ◁$directory_1$▷ ◁$directory_2$▷
  - Copia il contenuto di una directory in un'altra directory:
    `cp -r` ◁$directory_1$▷`/*` ◁$directory_2$▷
    dove $directory_1$ può essere omessa se è la directory di lavoro.
  - Ridenomina una directory:
    `mv` ◁$directory_1$▷ ◁$directory_2$▷
  - Cancella una directory:
    `rmdir` ◁*directory*▷ (se vuota)
    `rm -r` ◁*directory*▷ (se non vuota)
  - Accorpa una directory in un singolo file:
    `tar -cvf` ◁*file*▷`.tar` ◁*directory*▷
  - Accorpa il contenuto di una directory in un singolo file:
    `tar -cvf` ◁*file*▷`.tar` ◁*directory*▷`/*`
    dove *directory* può essere omessa se è la directory di lavoro.
  - Estrai ciò che è stato precedentemente accorpato in un unico file:
    `tar -xvf` ◁*file*▷`.tar`
  - Accorpa e comprimi una directory in un singolo file:
    `zip -r` ◁*file*▷`.zip` ◁*directory*▷
  - Accorpa e comprimi il contenuto di una directory in un singolo file:
    `zip -r` ◁*file*▷`.zip` ◁*directory*▷`/*`
    dove *directory* può essere omessa se è la directory di lavoro.
  - Decomprimi ed estrai ciò che è stato precedentemente accorpato e compresso in un unico file:
    `unzip` ◁*file*▷`.zip`

- Comandi relativi ai file:
  - Visualizza il contenuto di un file di testo o sorgente:
    `cat ◁`*file*`▷`
    Se il contenuto del file non sta tutto in una schermata, aggiungere il seguente filtro al precedente comando:
    `| more`
    al fine di visualizzare una schermata per volta.
  - Visualizza un file contenente un documento in formato PostScript:
    `gv ◁`*file*`▷.ps &`
  - Visualizza un file contenente un documento in formato PDF:
    `acroread ◁`*file*`▷.pdf &`
  - Trasforma un file di testo o sorgente in formato PostScript:
    `enscript -B -o ◁`$file_2$`▷.ps ◁`$file_1$`▷`
    e poi in formato PDF:
    `ps2pdf ◁`$file_2$`▷.ps ◁`$file_2$`▷.pdf`
  - Stampa un file contenente un documento in formato PostScript o PDF:
    `lpr -P◁`*stampante*`▷ ◁`*file*`▷`
  - Copia un file in un altro file (eventualmente appartenente a un'altra directory):
    `cp ◁`$file_1$`▷ ◁`$file_2$`▷`
  - Ridenomina un file (eventualmente spostandolo in un'altra directory):
    `mv ◁`$file_1$`▷ ◁`$file_2$`▷`
  - Cancella un file:
    `rm ◁`*file*`▷`
  - Comprimi il contenuto di un file:
    `gzip ◁`*file*`▷` oppure `zip ◁`*file*`▷.zip ◁`*file*`▷`
  - Decomprimi il contenuto di un file precedentemente compresso:
    `gunzip ◁`*file*`▷.gz` oppure `unzip ◁`*file*`▷.zip`

- Quando un file o una directory si trova su chiavetta USB, occorre procedere nel seguente modo:
  inserire la chiavetta USB nella porta USB
  `mount /pendrive`
  effettuare le operazioni desiderate attraverso la directory `/pendrive`
  `umount /pendrive`
  estrarre la chiavetta USB dalla porta USB

- I seguenti comandi servono per modificare la proprietà e i diritti di accesso relativi a file e directory:
  `chown ◁`*identificativo nuovo utente proprietario*`▷ ◁`*file o directory*`▷`
  `chgrp ◁`*identificativo nuovo gruppo proprietario*`▷ ◁`*file o directory*`▷`
  `chmod ◁`*nuovi diritti di accesso*`▷ ◁`*file o directory*`▷`
  dove i nuovi diritti di accesso vanno espressi attraverso tre cifre ottali corrispondenti alle tre triplette binarie.

- Il seguente comando è consigliabile per proteggere il contenuto della home directory di un utente in un sistema multiutente:
  `chmod 700 ~`
  dove la tripletta ottale `700` corrisponde alle tre triplette binarie `111 000 000` che stanno per `rwx --- ---`, ovvero attribuisce tutti i diritti di accesso all'utente proprietario e nessun diritto di accesso agli altri utenti.

- Tutti i suddetti comandi disponibili in modalità terminale, nonché i nomi dei file o directory a cui i comandi sono applicati, sono completabili attraverso il tabulatore, cioè basta scriverne la prima lettera o le prime lettere e poi premere il tabulatore per completarli a linea di comando. Inoltre il tasto freccia verso l'alto consente di richiamare in ordine cronologico inverso i comandi dati in precedenza.

■flbpp_1

8.3 L'Editor gvim

- Un file sorgente C, così come un file di testo, può essere creato in Linux attraverso il seguente comando:
  `gvim` ◁*file*▷
  dove `gvim` è un editor di testi piuttosto rapido, che evidenzia la sintassi in base all'estensione del file. Molti dei suoi comandi sono presenti anche nelle sue precedenti versioni non grafiche `vim` e `vi`, dove quest'ultimo è l'editor nativo di Unix e Linux ed è l'unico disponibile quando si deve lavorare su un dispositivo a livello sistemistico.
- Comandi di `gvim` per iniziare o terminare l'inserimento di caratteri in un file:
  - Entra in modalità inserimento testo nel punto in cui si trova il cursore oppure nel punto successivo:
    `i` oppure `a`
  - Entra in modalità inserimento testo all'inizio oppure alla fine della linea su cui si trova il cursore:
    `I` oppure `A`
  - Entra in modalità inserimento testo aprendo una nuova linea sotto oppure sopra la linea su cui si trova il cursore:
    `o` oppure `O`
  - Esci dalla modalità inserimento testo:
    `<esc>`
- Comandi di `gvim` per modificare rapidamente un file (fuori dalla modalità inserimento testo):
  - Sostituisci n caratteri consecutivi con n occorrenze di un nuovo carattere ($n = 1$ se omesso) a partire dal carattere su cui si trova il cursore:
    ◁*n*▷ `r` ◁*nuovo carattere*▷
  - Sostituisci n parole consecutive ($n = 1$ se omesso) con una sequenza di nuove parole a partire dal carattere su cui si trova il cursore:
    ◁*n*▷ `cw` ◁*nuove parole*▷ `<esc>`
  - Cancella n caratteri consecutivi ($n = 1$ se omesso) a partire dal carattere su cui si trova il cursore:
    ◁*n*▷ `x`
  - Cancella n parole consecutive ($n = 1$ se omesso) a partire dal carattere su cui si trova il cursore:
    ◁*n*▷ `dw`
  - Cancella n linee consecutive ($n = 1$ se omesso) a partire dalla linea su cui si trova il cursore:
    ◁*n*▷ `dd`
  - Copia n linee consecutive ($n = 1$ se omesso) a partire dalla linea su cui si trova il cursore:
    ◁*n*▷ `Y`
  - Incolla sotto oppure sopra la linea su cui si trova il cursore l'ultima sequenza di caratteri cancellati, l'ultima sequenza di parole cancellate o l'ultima sequenza di linee cancellate o copiate con `Y`:
    `p` oppure `P`
  - Ripeti l'ultimo comando tra quelli elencati sopra più `i`, `a`, `I`, `A`, `o`, `O`:
    `.`
  - Inserisci il contenuto di un altro file sotto la linea su cui si trova il cursore:
    `:r` ◁*file*▷ `<ret>`
  - Annulla l'ultimo comando tra quelli elencati sopra più `i`, `a`, `I`, `A`, `o`, `O`:
    `u`
- Comandi di `gvim` per salvare o scartare le ultime modifiche (fuori dalla modalità inserimento testo):
  - Salva le ultime modifiche:
    `:w <ret>`
  - Esci da `gvim`:
    `:q <ret>`
  - Salva le ultime modifiche ed esci da `gvim`:
    `:wq <ret>`
  - Esci da `gvim` senza salvare le ultime modifiche:
    `:q! <ret>`

- Comandi di `gvim` per muoversi all'interno di un file più celermente che con i tasti freccia (fuori dalla modalità inserimento testo):
  - Vai alla prima occorrenza di una stringa a partire dall'attuale posizione del cursore:
    `/◁`*`stringa`*`▷ <ret>`
    e poi per cercare una alla volta tutte le successive occorrenze:
    `/ <ret>`
  - Vai alla prima linea del file:
    `1G`
  - Vai all'ultima linea del file:
    `G`
  - Vai alla k-esima linea del file:
    `◁`*`k`*`▷G` oppure `:◁`*`k`*`▷ <ret>`
  - Spostati indietro di una parola rispetto all'attuale posizione del cursore:
    `b`
  - Spostati avanti di una parola rispetto all'attuale posizione del cursore:
    `e`
  - Spostati indietro di mezza schermata rispetto all'attuale posizione del cursore:
    `<ctrl>u`
  - Spostati avanti di mezza schermata rispetto all'attuale posizione del cursore:
    `<ctrl>d`
  - Spostati indietro di una schermata rispetto all'attuale posizione del cursore:
    `<ctrl>b`
  - Spostati avanti di una schermata rispetto all'attuale posizione del cursore:
    `<ctrl>f`
- Per impostare le opzioni di `gvim`, bisogna accedere ai file di configurazione `.vimrc` e `.gvimrc` nella propria home directory e scrivere rispettivamente in essi con `gvim` comandi come i seguenti:

```
set gfn=Courier\ 10\ Pitch\ 11          set guiheadroom=50
set lines=30                            colorscheme darkblue
set window=29
set columns=109
set textwidth=108
set ignorecase
set incsearch
set hlsearch
syntax on
```

  Il comando di `gvim` per controllare il valore di un'opzione e il file in cui è impostata è il seguente:
  `:verbose set ◁`*`opzione`*`▷?`
- Alcuni suggerimenti relativi allo stile di programmazione quando si scrive un file sorgente C:
  - Usare commenti per documentare lo scopo del programma: breve descrizione, nomi degli autori e loro affiliazioni, numero e data di rilascio della versione corrente, modifiche apportate nelle versioni successive alla prima.
    Usare commenti per documentare lo scopo delle costanti simboliche, dei nuovi tipi, delle definizioni di funzioni, dei parametri formali, delle variabili locali e dei gruppi di istruzioni correlate.
    Riportare come commenti le considerazioni effettuate durante l'analisi del problema (dati di ingresso e uscita e loro relazioni) e i passi principali individuati durante la progettazione dell'algoritmo.
  - Lasciare alcune righe vuote tra le inclusioni di librerie e le definizioni di costanti simboliche, tra queste ultime e le definizioni di tipi, tra queste ultime e le dichiarazioni di funzioni, tra queste ultime e le definizioni di funzioni, e tra due definizioni consecutive di funzioni.
    All'interno della definizione di una funzione, lasciare una riga vuota tra la sequenza di dichiarazioni di variabili locali e la sequenza di istruzioni.
    All'interno della sequenza di istruzioni di una funzione, lasciare una riga vuota tra due sottosequenze consecutive di istruzioni logicamente correlate.

  - Indentare il corpo di ciascuna funzione di almeno due caratteri rispetto a "`{`" e "`}`".
    Indentare opportunamente le istruzioni di controllo del flusso `if`, `switch`, `while`, `for` e `do-while`.
  - Disporre su più righe e allineare gli identificatori di variabili dichiarati dello stesso tipo.
    Disporre su più righe e allineare le dichiarazioni dei parametri formali delle funzioni.
    Disporre su più righe e allineare i parametri effettivi contenuti nelle invocazioni delle funzioni.
  - Un commento può estendersi su più righe e può comparire da solo o al termine di una direttiva, dichiarazione o istruzione. Se esteso su più righe, non deve compromettere l'indentazione.
    Una dichiarazione o istruzione può estendersi su più righe a patto di non andare a capo all'interno di un identificatore o una costante letterale. Se estesa su più righe, non deve compromettere l'indentazione.
  - Per gli identificatori introdotti da chi programma, si rimanda alla Sez. 2.5. È inoltre consigliabile che essi siano tutti espressi nella stessa lingua.
  - Lasciare uno spazio vuoto prima e dopo ogni operatore binario.
    Lasciare uno spazio vuoto dopo `if`, `else`, `switch`, `case`, `while`, `for` e `do`.
    Non lasciare nessuno spazio vuoto tra l'identificatore di una funzione e "`(`".
    Non lasciare nessuno spazio vuoto dopo "`(`" e prima di "`)`".
    Non lasciare nessuno spazio vuoto prima di "`,`", "`;`", "`?`" e "`:`".

- Esercizio: Creare una directory chiamata `conversione_mi_km` e scrivere al suo interno con `gvim` un file chiamato `conversione_mi_km.c` per il programma di Sez. 2.2.

- Esercizio: Creare una directory chiamata `conversione_mi_km_file` e scrivere al suo interno con `gvim` un file chiamato `conversione_mi_km_file.c` per il programma di Sez. 2.8. ■flbpp_2

8.4 Il Compilatore gcc

- Un programma C può essere reso eseguibile in Linux attraverso il compilatore `gcc`, che è una variante del compilatore `cc` nativo di Unix e Linux.

- Comando per compilare un programma C scritto su un singolo file sorgente:
  `gcc -ansi -Wall -O ◁`*file sorgente*`▷.c -o ◁`*file eseguibile*`▷`
  dove:
  - L'opzione `-ansi` impone al compilatore di controllare che il programma rispetti lo standard ANSI.
  - L'opzione `-Wall` impone al compilatore di riportare tutti i messaggi di warning (potenziali fonti di errore, come ad esempio l'uso di una variabile non inizializzata nel lato destro di un assegnamento).
  - L'opzione `-O` impone al compilatore di ottimizzare il file eseguibile.
  - L'opzione `-o` permette di dare al file eseguibile un nome diverso da quello di default `a.out`. È opportuno dare al file eseguibile lo stesso nome, a meno dell'estensione `.c`, del file sorgente.

- Sequenza di comandi per compilare un programma C scritto su $n \geq 2$ file sorgenti (di cui solo il primo contiene la funzione `main` mentre gli altri sono i file di implementazione di altrettante librerie):
  `gcc -ansi -Wall -O -c ◁`$file_1$`▷.c`
  `gcc -ansi -Wall -O -c ◁`$file_2$`▷.c`
  ⋮
  `gcc -ansi -Wall -O -c ◁`$file_n$`▷.c`
  `gcc -ansi -Wall -O ◁`$file_1$`▷.o ◁`$file_2$`▷.o ... ◁`$file_n$`▷.o -o ◁`*file eseguibile*`▷`
  dove:
  - L'opzione `-c` impone al compilatore di produrre un file oggetto (anziché un file eseguibile) avente lo stesso nome del file sorgente ed estensione `.o`.
  - L'ultimo comando crea un file eseguibile collegando i file oggetto precedentemente ottenuti.

- Se il programma include il file di intestazione della libreria standard `math.h`, potrebbe rendersi necessario aggiungere l'opzione `-lm` in fondo al comando `gcc`.
- Il file eseguibile viene prodotto solo se il compilatore non riscontra:
  - Errori lessicali: violazioni del lessico del linguaggio.
  - Errori sintattici: violazioni delle regole grammaticali del linguaggio.
  - Errori semantici: violazioni del sistema di tipi del linguaggio.
  - Errori di collegamento: utilizzo in un file oggetto di un identificatore non locale la cui definizione o dichiarazione non si trova in nessuno degli altri file oggetto.
- Quando rileva un errore, il compilatore emette un messaggio che indica linea e carattere dove si trova l'errore nel file sorgente (in realtà potrebbe trovarsi un po' più indietro) e il genere di errore incontrato. In base ai messaggi d'errore occorre modificare il programma e poi ricompilarlo. È raccomandabile modificare e ricompilare il programma anche in caso di segnalazioni di warning, cioè situazioni come l'uso di una variabile locale non inizializzata che non impediscono il completamento della compilazione.
- Comando per lanciare in esecuzione il file eseguibile di un programma C:
  ./◁file eseguibile▷
- Se si compiono preventivamente operazioni come le seguenti:
  `gvim ~/.cshrc`
  cambiare la linea contenente la definizione di `path` come segue:
  `set path = ($path .)`
  uscire da `gvim`
  `source ~/.cshrc`
  allora il comando per lanciare in esecuzione il file eseguibile di un programma C si semplifica come segue:
  ◁file eseguibile▷

8.5 L'Utility di Manutenzione `make`

- È buona norma raccogliere tutti i file sorgenti che compongono un programma – ad eccezione dei file di libreria standard – in una directory, eventualmente organizzata in sottodirectory al suo interno. Per la manutenzione del contenuto di tale directory si può ricorrere in Linux all'utility `make`, la quale esegue i comandi specificati in un file di testo chiamato `Makefile` da creare all'interno della directory stessa.
- Il `Makefile` (o `makefile`) è una sequenza di direttive, ciascuna della seguente forma:
  #◁eventuale commento▷
  ◁obiettivo▷: ◁eventuale lista delle dipendenze▷
  ◁azione▷
  dove:
  - L'obiettivo è il nome della direttiva e nella maggior parte dei casi coincide con il nome di un file che si vuole produrre. La sintassi del comando `make` è di conseguenza la seguente:
    `make` *◁obiettivo▷*
    dove *obiettivo* può essere omesso se è il primo della lista all'interno del `Makefile`.
  - Le dipendenze rappresentano i file da cui dipende il conseguimento dell'obiettivo. Se anche uno solo di questi file non esiste e non è specificata all'interno del `Makefile` una direttiva che stabilisce come ottenerlo, allora l'obiettivo non può essere raggiunto e l'esecuzione di `make` termina segnalando un errore.
  - L'azione, che deve essere preceduta da un carattere di tabulazione, rappresenta una sequenza di comandi che l'utility `make` deve eseguire per raggiungere l'obiettivo a partire dai file da cui l'obiettivo dipende. Se l'obiettivo è un file esistente, l'azione viene eseguita solo se almeno uno dei file da cui l'obiettivo dipende ha una data di ultima modifica successiva alla data di ultima modifica del file obiettivo, così da evitare lavoro inutile.

- Nel caso della compilazione, l'obiettivo è il file eseguibile, il suo ottenimento dipende dalla disponibilità dei file sorgente e l'azione per costruirlo è il comando `gcc`.
- Esempi:
  - `Makefile` per la compilazione di un programma C scritto su un singolo file sorgente:

```
◁file eseguibile▷:  ◁file sorgente▷.c Makefile
        gcc -ansi -Wall -O ◁file sorgente▷.c -o ◁file eseguibile▷
pulisci:
        rm -f ◁file sorgente▷.o
pulisci_tutto:
        rm -f ◁file eseguibile▷ ◁file sorgente▷.o
```

  - `Makefile` per la compilazione di un programma C scritto su $n \geq 2$ file sorgenti:

```
◁file eseguibile▷:  ◁file1▷.o ◁file2▷.o ... ◁filen▷.o Makefile
        gcc -ansi -Wall -O ◁file1▷.o ◁file2▷.o ... ◁filen▷.o -o ◁file eseguibile▷
◁file1▷.o:  ◁file1▷.c ◁file di intestazione di librerie non standard inclusi▷ Makefile
        gcc -ansi -Wall -O -c ◁file1▷.c
◁file2▷.o:  ◁file2▷.c ◁file di intestazione di librerie non standard inclusi▷ Makefile
        gcc -ansi -Wall -O -c ◁file2▷.c
  ⋮
◁filen▷.o:  ◁filen▷.c ◁file di intestazione di librerie non standard inclusi▷ Makefile
        gcc -ansi -Wall -O -c ◁filen▷.c
pulisci:
        rm -f *.o
pulisci_tutto:
        rm -f ◁file eseguibile▷ *.o
```

    dove `*.o` significa tutti i file il cui nome termina con `.o`.
- Esercizio: Nella directory `conversione_mi_km` scrivere con `gvim` un `Makefile` per la compilazione con `gcc` di `conversione_mi_km.c`, eseguire il `Makefile` e lanciare in esecuzione il file eseguibile ottenuto al fine di testarne il corretto funzionamento.
- Esercizio: Nella directory `conversione_mi_km_file` scrivere con `gvim` un `Makefile` per la compilazione con `gcc` di `conversione_mi_km_file.c`, eseguire il `Makefile` e lanciare in esecuzione il file eseguibile ottenuto al fine di testarne il corretto funzionamento. Prima di ogni esecuzione, nella stessa directory occorre preventivamente scrivere con `gvim` un file chiamato `miglia.txt` da cui il programma acquisirà il valore da convertire. Al termine di ogni esecuzione, il risultato della conversione si troverà in un file chiamato `chilometri.txt` creato dal programma nella stessa directory. ■flbpp_3

8.6 Il Debugger `gdb`

- Raramente un programma compilato con successo produce i risultati attesi le prime volte che viene eseguito. In particolare, possono verificarsi i seguenti errori d'esecuzione:
  - Ottenimento di risultati diversi da quelli attesi, a causa del fatto che il programma non è stato progettato correttamente rispetto al problema assegnato oppure della mancata comprensione di alcuni aspetti tecnici del linguaggio di programmazione utilizzato.
  - Mancato ottenimento di risultati, a causa della non terminazione dell'esecuzione del programma oppure della terminazione prematura dell'esecuzione perché il programma ha indotto il computer a svolgere operazioni illegali che il sistema operativo ha rilevato (p.e. l'uso di una variabile di tipo puntatore che vale `NULL` oppure il calcolo di una divisione per zero).
- Per ridurre la possibilità che si verifichino errori d'esecuzione occorre pianificare adeguate attività di verifica a priori e testing a posteriori (vedi Sez. 7.1).

- Quando si verifica un errore d'esecuzione, non si ha a disposizione uno strumento come un compilatore, che localizza gli errori linguistici automaticamente, ma bisogna condurre una vera e propria indagine. Questa ha luogo tramite un debugger, il quale ci permette di eseguire il programma passo passo, di ispezionare il valore delle variabili come pure il contenuto dello stack dei record di attivazione, e di impostare dei punti di arresto (breakpoint) in corrispondenza di determinate parti del programma.
- Comando per lanciare in esecuzione il debugger `gdb` su un programma C:
  `gdb` ◁*file eseguibile*▷
- L'utilizzo di `gdb` richiede che il programma sia stato compilato specificando anche l'opzione `-g` ogni volta che `gcc` è stato impiegato. L'opzione `-g`, che è incompatibile con l'opzione `-O` e quindi va usata al suo posto, arricchisce il file eseguibile con le metainformazioni di cui `gdb` necessita per svolgere i compiti precedentemente indicati.
- Questi sono alcuni dei comandi di più frequente utilizzo disponibili in `gdb`:
  - Avvia l'esecuzione del programma con l'assistenza di `gdb`:
    `run`
    L'esecuzione si arresta al primo breakpoint, al primo errore che causa la terminazione prematura dell'esecuzione, oppure a seguito di normale terminazione (giusti o sbagliati che siano i risultati).
  - Visualizza il contenuto dello stack dei record di attivazione in termini di chiamate di funzione in corso di esecuzione:
    `bt`
    Questo è il primo comando da dare al verificarsi di un errore d'esecuzione.
  - Visualizza il valore di un'espressione C:
    `print` ◁*espressione C*▷
    dove l'espressione può contenere solo identificatori visibili nel punto in cui il comando viene dato (come identificatori di parametri formali e variabili locali della funzione in corso di esecuzione). Ciò è particolarmente utile per conoscere il contenuto di parametri e variabili durante l'esecuzione.
  - Imposta un breakpoint all'inizio di una funzione:
    `break` ◁*funzione*▷
    oppure presso una linea di un file sorgente:
    `break` ◁*file sorgente:*▷ ◁*numero linea*▷
    dove l'indicazione del file sorgente può essere omessa se c'è un unico file sorgente. A ogni breakpoint impostato viene automaticamente associato un numero seriale univoco.
  - Imponi una condizione espressa in C al verificarsi della quale l'esecuzione deve arrestarsi presso un breakpoint precedentemente impostato:
    `condition` ◁*numero seriale breakpoint*▷ ◁*espressione C*▷
    dove l'espressione non deve contenere identificatori non visibili nel punto di arresto. Se l'espressione viene omessa si impone un arresto incondizionato presso il breakpoint, cancellando così l'eventuale condizione definita in precedenza per lo stesso breakpoint.
  - Esegui la prossima istruzione senza entrare nell'esecuzione passo passo delle funzioni da essa invocate:
    `next`
    oppure:
    `n`
  - Esegui la prossima istruzione entrando nell'esecuzione passo passo delle funzioni da essa invocate:
    `step`
    oppure:
    `s`
  - Continua l'esecuzione fino a incontrare il prossimo breakpoint:
    `continue`
    oppure:
    `c`

  - Elimina un breakpoint:
    `clear` ◁*funzione*▷
    oppure:
    `clear` ◁*file sorgente:*▷ ◁*numero linea*▷
    oppure:
    `delete` ◁*numero seriale breakpoint*▷
  - Ottieni informazioni sui comandi di `gdb`:
    `help`
    oppure:
    `apropos`
  - Esci da `gdb`:
    `quit`
    oppure:
    `q`

- L'istruzione che viene mostrata in corrispondenza di un breakpoint è la prossima istruzione da eseguire. Se si tratta di un assegnamento, prima di poter ispezionare il nuovo valore della variabile a sinistra dell'operatore di assegnamento bisogna avanzare col comando `n`. Nel caso di un breakpoint relativo a una funzione, i valori dei parametri di quella funzione sono invece già ispezionabili.
- Se il programma deve essere lanciato in esecuzione con dei parametri a linea di comando, che verranno gestiti dalla funzione `main` tramite i suoi parametri formali `argc` e `argv`, allora quei parametri devono seguire ogni utilizzo del comando `run` nel seguente modo:
  `run` ◁*parametri effettivi del programma*▷
  In alternativa, si possono impostare quei parametri con:
  `set args` ◁*parametri effettivi del programma*▷
  il che consente di richiamare poi il comando `run` più volte sugli stessi valori di quei parametri.
- Se `gdb` viene avviato nel seguente modo:
  `gdb -tui` ◁*file eseguibile*▷
  allora parte dello schermo viene riservato a mostrare l'avanzamento dell'esecuzione sul file sorgente.
- Mentre il debugging ordinario formula ipotesi sulle cause e poi procede all'avanti per riprodurre l'errore, il debugging reversibile parte dagli effetti dell'errore e poi risale alle cause. Per abilitare quest'ultimo, occorre usare l'opzione `-static` durante la compilazione con `gcc` e poi dare i seguenti comandi in `gdb`:
  ```
  break _start
  run
  record
  continue
  ```
  Ecco alcuni comandi di debugging reversibile disponibili in `gdb`:
  ```
  reverse-next
  reverse-step
  reverse-continue
  ```
- Esercizio: Nella directory `conversione_mi_km` lanciare in esecuzione il programma e verificarne il comportamento nel caso in cui come distanza in miglia si introduca un valore non numerico. Modificare poi `conversione_mi_km.c` con `gvim` per aggiungere la validazione lasca dell'input (vedi Sez. 4.4), rieffettuare la compilazione tramite il `Makefile`, rieseguire il programma per verificarne di nuovo il comportamento nello stesso caso di prima e utilizzare `gdb` per comprendere la necessità di ricorrere in quel caso alla validazione stretta dell'input (vedi Sez. 4.4) ispezionando `*stdin`.
- Esercizio: Nella directory `conversione_mi_km_file` lanciare in esecuzione il programma e verificarne il comportamento nel caso in cui il file `miglia.txt` non sia presente nella directory e nel caso in cui tale file contenga come distanza in miglia un valore non numerico. Modificare poi `conversione_mi_km_file.c` con `gvim` per aggiungere il controllo di errata apertura di file in lettura e la validazione stretta dell'input da file (se il valore contenuto nel file di input è errato, scrivere un messaggio di errore nel file di output), rieffettuare la compilazione tramite il `Makefile` e rieseguire il programma per verificarne di nuovo il comportamento negli stessi due casi di prima. ■flbpp_4

8.7 Implementazione e Modifica di Programmi e Librerie C

- Esercizi: Per ciascuno dei seguenti esempi di programma/libreria introdotti durante le lezioni teoriche, creare una nuova directory in cui scrivere usando `gvim` i file sorgenti (completi di validazione stretta di tutti i valori acquisiti in ingresso – vedi Sez. 4.4) e il `Makefile` per la loro compilazione con `gcc`, eseguire il `Makefile` e lanciare in esecuzione il file eseguibile ottenuto al fine di testarne il corretto funzionamento (avvalendosi di `gdb` ogni volta che si verificano errori d'esecuzione):

  1. Programma per il calcolo della bolletta dell'acqua (Sez. 4.3).
  2. Programma per il calcolo dei livelli di radiazione (Sez. 4.4).
  3. Libreria per l'aritmetica con le frazioni (Sezz. 5.10 e 5.7).
  4. Libreria per le operazioni matematiche ricorsive sui numeri naturali (Sez. 5.8).
  5. Programma per la determinazione del valore di un insieme di monete (Sez. 3.11) modificato facendo uso di istruzioni di ripetizione (Sez. 4.4) e di strutture dati di tipo array (Sez. 6.8) e di tipo stringa (Sez. 6.9) al fine di evitare le ridondanze di codice presenti nella versione originale del programma.
  6. Programma per la statistica delle vendite (Sez. 6.8) modificato con la validazione dei dati del file.
  7. Libreria per la gestione delle figure geometriche (Sez. 6.10).
  8. Libreria per la gestione delle liste ordinate (Sez. 6.11).

  Si rammenta che per ogni programma che fa uso di file è anche necessario scrivere con `gvim` i relativi file di input prima che il programma venga eseguito. Si ricorda inoltre che per ogni libreria è pure necessario scrivere con `gvim` un file sorgente che include il file di intestazione della libreria, definisce la funzione `main` e fa uso di tutte le funzioni esportate dalla libreria. ■fegpp_1–12