Appunti di Informatica, volume 1

Alessandro Bugatti

18 settembre 2024

Alessandro Bugatti

Alla mia famiglia.

– Alessandro

ISBN: 978-1-326-97439-8

Immagine in copertina: Chiara Bugatti, *Albero della stanza*, tecnica mista, presente solo nella versione a stampa.

Prefazione

Lo scopo di questi appunti è di fornire un percorso didattico che rifletta quella che è la mia esperienza di insegnamento, favorendo l'aspetto didattico rispetto a quello strettamente tecnico, senza comunque sacrificare la precisione.

Ho quindi scelto di privilegiare gli argomenti e le modalità che costituiscono quella che, secondo me, è l'ossatura di una buona preparazione informatica, favorendo i concetti fondamentali rispetto alle tecnologie o ai linguaggi che, mutando rapidamente, risultano meno rilevanti per porre delle solide basi. Mi ritrovo molto nell'analisi fatta da *Frederick P. Brooks, Jr.* nel famoso articolo «No Silver Bullet - Essence and Accident in Software Engineering» del 1986, in cui la costruzione di software viene divisa tra parti *essenziali* e parti *accidentali*, dove le prime sono inerenti al processo stesso, e quindi in qualche modo *senza tempo*, mentre le seconde sono il risultato di alcune scelte tecnologiche e come tali possono cambiare anche in maniera piuttosto frequente.

Per fare un esempio, una parte essenziale nella costruzione/progettazione/implementazione di un software riguarda la complessità del problema che si vuole risolvere e le relazioni, spesso intricate, tra le varie parti della soluzione; una parte accidentale è invece l'utilizzo di un particolare linguaggio di programmazione o di un particolare editor.

Ovviamente è stata fatta una scelta di strumenti e tecnologie, ma non perché in qualche modo questi costituiscano i *"Best programming languages to learn in 20XX"*, ma perché sono quelli che conosco meglio e su cui posso fare delle scelte sensate riguardo cosa trattare e cosa no.

Ritengo che l'insegnamento dell'informatica, o forse l'insegnamento in generale, debba passare attraverso la sensibilità dell'insegnante rispetto al soggetto dell'insegnamento, che quindi deve essere plasmato ed adattato a ciò che si ritiene importante trasmettere ai propri studenti.

In questo volume si affronta la tematica generale di cosa voglia dire *programmare*, partendo dai concetti di base e utilizzando il C++ come linguaggio per la scrittura degli algoritmi che verranno di volta in volta proposti.

Prefazione all'edizione 2024

Questa è la prima edizione che assume la forma di un testo completo: la sua produzione ha richiesto innumerevoli ore di scrittura, più alcune dedicate alla rilettura e alla rilevazione di errori. Non essendoci però un processo di vero e proprio *editing* suppongo che possano essere presenti ancora molti errori, sia testuali che tipografici (spero molto pochi nei contenuti): l'idea è quello di procedere con una revisione costante, quindi questo testo può essere visto come una «rolling release» che migliorerà con il tempo e con l'uso.

Buono studio e buona lettura.

Indice

1. Informatica e algoritmi

Introduzione

L'*informatica*[1] è una scienza, nata con l'avvento dei computer, che si occupa del trattamento dell'informazione attraverso l'utilizzo di procedure automatiche. La parola stessa ha come radice il termine *informazione* a cui viene aggiunto il suffisso *-ica* per indicare il fatto che sia la *scienza dell'informazione*, come avviene per altre scienze come la matematica, la linguistica, l'ottica, ecc.

L'informazione è un qualsiasi dato che, all'interno di un contesto, permette di aggiungere un nuovo pezzo di conoscenza alla propria esperienza del mondo e che può essere comunicata, memorizzata, elaborata e altro.

Vediamo alcuni esempi di trattamento dell'informazione:

- uno studente si iscrive a scuola e **comunica** tutte le informazioni personali come nome, cognome, data di nascita, cittadinanza, ecc., che vengono inserite all'interno dei software amministrativi e didattici della scuola, come il registro elettronico, in modo che possano essere **memorizzate** per una successiva eventuale **elaborazione**. Successivamente queste informazioni continuano a essere arricchite, aggiungendone di nuove come voti, lavori svolti, assenze, ritardi e altro.
- un utente si iscrive a un *social network*, **comunicando** anche in questo caso alcune informazioni personali, che verranno **memorizzate** nei server dell'azienda che gestisce l'applicazione e successivamente **elaborate** per creare le associazioni tra gli utenti, aggiornarli sulle modifiche al proprio stato che vengono fatte dagli utenti con i quali esiste un collegamento, stabilire cosa possa interessare a ogni utente e tanto altro ancora.

L'aspetto più interessante che verrà approfondito nel corso del libro è quello relativo alla possibilità di elaborare l'informazione, che verrà fatta attraverso la creazione di algoritmi.

1: In inglese invece la parola più utilizzata è *computer science*, nei contesti dove noi solitamente utilizziamo informatica. Esiste anche la parola *informatics*, ma è riservata ad ambiti più specifici, ad esempio Olympiad in Informatics, Olimpiadi di Informatica.

1.1. Algoritmi

Per programmare un computer in modo che sia in grado di elaborare delle informazioni per risolvere un qualsiasi problema dato,

bisogna come prima cosa essere in grado di risolvere il problema da un punto di vista teorico, poiché la programmazione altro non è che "*istruire*" un computer in modo che possa compiere una serie di azioni al nostro posto, in maniera più rapida, senza annoiarsi e senza commettere errori.

È evidente che se il programmatore non è in grado di risolvere il problema, nemmeno il computer lo potrà fare, potrà al limite eseguire una serie di azioni che non permetteranno di arrivare alla soluzione. L'insieme di azioni che portano alla soluzione di un problema viene in genere chiamato **algoritmo**[2] e una definizione semplice e efficace potrebbe essere questa:

2: Il termine deriva dal nome del matematico persiano al-Khwarizmi, vissuto nel IX secolo d.C., successivamente adattato durante la diffusione delle sue opere in Europa.

Definizione di algoritmo

Un algoritmo è una sequenza finita di passi (istruzioni, azioni) elementari che, quando eseguiti, portano alla soluzione di un problema di carattere generale.

Analizzando questa definizione si possono notare una serie di cose interessanti:

- **una sequenza finita di passi**: quindi non un insieme in un ordine qualsiasi, ma una *sequenza*, cioè ogni istruzione deve essere nel posto giusto, con una precisa istruzione che la precede e una che la segue. Inoltre la sequenza deve essere *finita*, in modo da poter essere descritta.
- **elementari**: il concetto di elementare va riferito rispetto all'esecutore dell'algoritmo. Siccome per un programmatore l'esecutore di un algoritmo è il computer, allora elementari significa istruzioni che possono essere eseguite immediatamente dal computer (come ad esempio fare la somma di due numeri), mentre non sono elementari quelle istruzioni che per essere eseguite hanno bisogno di essere scomposte in istruzioni più semplici (come ad esempio calcolare l'area di un rettangolo)
- **quando eseguiti**: l'algoritmo è una descrizione statica della soluzione, deve essere eseguito per portare a una soluzione effettiva
- **portano alla soluzione**: sembra ovvio, ma uno dei requisiti di un algoritmo è che deve portare alla soluzione di un problema. Un algoritmo che non porti alla soluzione di un problema è evidentemente inutile.
- **di carattere generale**: un algoritmo dipende dai *dati in ingresso* e in questo modo può risolvere tutti i casi di un problema. Se ad esempio avessimo l'algoritmo per il calcolo dell'area di un rettangolo di base 3 e altezza 4 questo sarebbe utile solo nel caso specifico in cui interessi l'area di quel particolare

rettangolo, mentre è ovviamente molto più interessante avere un algoritmo che calcoli l'area di qualsiasi rettangolo, dati i valori della base e dell'altezza.

Dall'ultima caratteristica si può notare che, per ottenere la generalità di un algoritmo, è necessario avere dei dati in ingresso che verranno trasformati in dati in uscita. come si può vedere nella Figura 1.1.

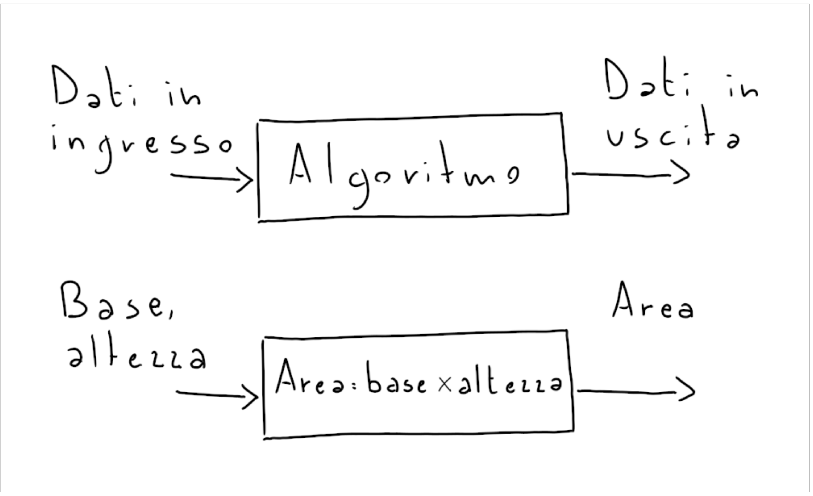


Figura 1.1.: Schema di funzionamento di un algoritmo

Anche se questo schema può sembrare a prima vista applicabile solo a problemi di tipo matematico, come quello mostrato relativo all'area del rettangolo, in realtà è in grado di rappresentare qualsiasi genere di problema, a patto di individuare correttamente quali sono i dati in ingresso e quali sono i dati in uscita.

Un esempio potrebbe essere quello della determinazione della collisione tra due oggetti all'interno di un videogioco, problema che deve essere risolto svariate volte al secondo e la cui soluzione implica ad esempio la possibilità che uno o entrambi gli oggetti spariscano dal gioco.

Esempio videogioco 2D

Problema: in un videogioco 2D si vuole sapere se un proiettile ha colpito il giocatore

Dati in ingresso: posizione del giocatore, posizione del proiettile, eventuali bounding box o altro

Dati in uscita: una risposta della forma vero/falso

Un altro esempio, legato ai *social network*, è quello di determinare gli amici di un certo utente, problema che in alcuni social network permette poi di mostrare all'utente aggiornamenti dello stato dei propri amici.

Amici in un social network

Problema: in un social network si vogliono sapere gli amici di un certo utente

Dati in ingresso: identificativo dell'utente

Dati in uscita: una lista di identificativi degli utenti che sono suoi amici

1.2. Diagrammi di flusso

Una volta definito il concetto di algoritmo, è necessario trovare un modo per descriverlo che possa essere utile per realizzare successivamente un programma per computer.

La descrizione più ovvia è quella in linguaggio *naturale*, che nel nostro caso è la lingua italiana. Si prenda il seguente come esempio di problema da risolvere tramite un algoritmo:

Esempio di problema

Problema: si vuole sapere quante sono le potenze del 2 comprese tra 1 e un dato numero **N** positivo, estremi inclusi.

Come prima cosa bisogna trovare un modo per risolverlo, quindi un *algoritmo risolutivo*: a titolo di esempio verranno forniti tre differenti modalità per risolvere questo problema, ognuna enunciata in linguaggio naturale.

Algoritmo 1

Si prenda ogni numero da 1 fino a N e si verifichi se è o meno una potenza di due: se lo è, si aumenti di un'unità il conteggio delle potenze del 2, altrimenti non si faccia niente.

Algoritmo 2

Si parta dal numero 1 e lo si raddoppi aumentando contemporaneamente il conteggio delle potenze del 2. Quando questo processo porta alla produzione di un numero maggiore di N ci si fermi e il valore del conteggio ottenuto è la soluzione.

Algoritmo 3

Si applichi la formula matematica[3]

$$soluzione = \lfloor log_2 N \rfloor + 1$$

Ognuna delle tre soluzioni proposte, come vedremo in seguito, porta al risultato corretto: il problema risiede nel fatto che il linguaggio naturale può avere diverse interpretazioni e, soprattutto, che il computer non è in grado di comprenderlo e trasformarlo in una sequenza di passaggi per arrivare alla soluzione[4].

Prima di affrontare i linguaggi per computer che servono a descrivere gli algoritmi, si utilizzerà uno strumento concettuale che permetterà di dare una forma più precisa alle idee risolutive. Questo strumento è il *diagramma di flusso*[5] o *flowchart* in inglese, termini che vengono usati in maniera intercambiabile. Il diagramma di flusso è un formalismo grafico che, attraverso delle forme geometriche, permette di descrivere in maniera precisa i vari passi che compongono un algoritmo.

Essendo stato proposto prima ancora dell'avvento dei computer in ambiti che non erano quelli della descrizione di algoritmi, quando si diffuse e iniziò a essere utilizzato per gli scopi che interessano a noi, si svilupparono diverse versioni che, pur condividendo le idee di base, aggiunsero o modificarono quella che era la rappresentazione originale. Dovendo riferirsi a una rappresentazione particolare, in questo testo verrà usata quella che può essere generata attraverso il software **Flowgorithm**, che può essere scaricato gratuitamente all'indirizzo `http://www.flowgorithm.org/`[6].

Inizio e fine di un algoritmo

Ogni algoritmo, essendo una sequenza finita di passi, avrà un *inizio* e una *fine*. Nei diagrammi di flusso i blocchi che rappresentano queste due situazioni sono i seguenti:

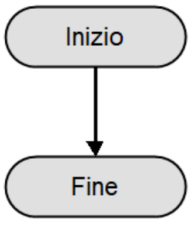


Figura 1.2.: Un algoritmo minimale con solo l'inizio e la fine

Come si può vedere sia l'inizio che la fine sono rappresentati all'interno di rettangoli con gli spigoli arrotondati, e in questo semplicissimo algoritmo, che non fa niente, vengono uniti da una freccia orientata dall'inizio verso la fine. Come si vedrà in seguito

3: A voler essere precisi una formula matematica non è linguaggio naturale, poichè usa una notazione e dei simboli che solo degli specialisti sono in grado di interpretare e quindi, per certi versi, non è molto differente dai linguaggi di programmazione che si andranno a studiare.

4: Questo non vuol dire che i computer non siano in grado di «comprendere» il linguaggio naturale, solo che in quel caso il problema è proprio la comprensione del testo, mentre nel caso che interessa a noi si vuole arrivare a una rappresentazione che sia quella *nativa* del computer e che quindi si limiti a eseguire i nostri comandi.

5: A volte definito anche diagrammi a blocchi per motivi che saranno chiari nel seguito.

6: Un altro sito che contiene molta documentazione ed esempi, anche in italiano, è quello di Roberto Atzori, che si trova all'indirizzo `http://flowgorithm.altervista.org`

quando i diagrammi diventeranno più complessi, il *flusso* è appunto rappresentato dalle frecce e dalla loro direzione.

Operazioni di lettura/scrittura

Le operazioni di lettura/scrittura rappresentano l'input/output del programma e vengono rappresentate nel modo seguente:

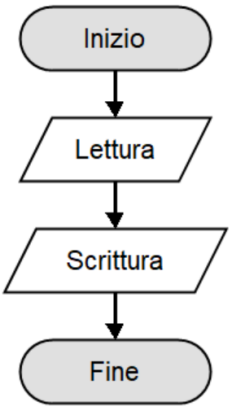


Figura 1.3.: Blocchi di lettura/scrittura rappresentati da parallelogrammi

Come si può vedere nella Figura 1.3, l'input[7] è l'insieme dei dati che l'utente fornisce a un algoritmo e l'output è l'insieme dei dati del l'algoritmo ritorna all'utente e che in generale rappresentano la soluzione del problema.

7: In Informatica ingresso, input e lettura vengono usate come sinomini, allo stesso modo con cui succede per uscita, output e scrittura.

Nell'esempio di Figura 1.1 l'input è rappresentato dalla base e dall'altezza di un rettangolo, mentre l'area è l'output, cioè la soluzione del problema. Nella figura Figura 1.3 però non sono presenti nessuno di questi elementi, quindi per poter indicare in modo specifico cosa sia l'input e cosa l'output si ricorre all'utilizzo di *variabili*, il cui significato verrà spiegato nel prossimo paragrafo e ripreso e approfondito nel prossimo capitolo.

Utilizzando le variabili, il diagramma della Figura 1.3 diventerà:

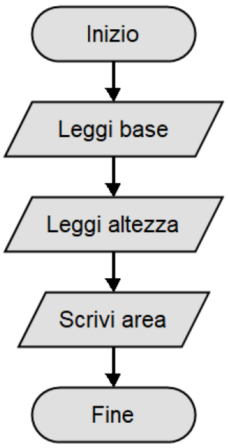


Figura 1.4.: Nei blocchi di lettura e scrittura vengono indicati quali variabili leggere/scrivere

dove in input (lettura) vengono acquisiti i valori di base e altezza, mentre in output (scrittura) viene mostrato il valore dell'area calcolata dall'algoritmo risolutivo.

Dichiarazione delle variabili

Le variabili altro non sono che dei "contenitori" di valori: sempre facendo riferimento all'esempio del rettangolo, la variabile base potrebbe contenere il valore 5, piuttosto che 7.4 o 19 o qualsiasi altro valore, e l'algoritmo utilizzerebbe quel valore per produrre il risultato corretto.

In alcuni linguaggi di programmazione non è necessario *dichiarare* le variabili, nel momento stesso in cui si utilizzano vengono "create", invece in Flowgorithm e nel linguaggio che verrà introdotto in seguito, il C++, le variabili hanno bisogno di essere dichiarate prima di poter essere utilizzate.

La *dichiarazione* di una variabile può essere definita in questo modo

Dichiarazione di una variabile

Dichiarare una variabile significa far conoscere all'ambiente di svolgimento dell'algoritmo due aspetti fondamentali:

- il **nome** con il quale verrà utilizzata nell'algoritmo
- il **tipo** della variabile: per tipo si intende cosa potrà contenere come valore la variabile, ad esempio un numero, una parola, una data o altro

Per quanto riguarda il nome, non tutti i nomi possono essere validi[8], per il momento si sceglieranno dei nomi che al loro interno contengano solo caratteri minuscoli o maiuscoli ed eventualmente cifre in fondo al nome[9]. Esempi di nomi validi sono **base**, **n1**, **tempoFrenata**, nomi non validi sono **1n**, **Base_rettangolo**, **tempo frenata** con uno spazio tra tempo e frenata.

La scelta del nome di una variabile riveste particolare importanza nella progettazione di un algoritmo: al momento basti sapere che deve essere significativo rispetto al problema che si intende risolvere, negli esempi e esercizi successivi verrà mostrato cosa si intende esattamente.

Per quanto riguarda il tipo, in questo capitolo verranno usate solo variabili di tipo numerico, in particolare numeri interi oppure numeri con la virgola[10].

Riprendendo l'esempio del rettangolo, la dichiarazione delle variabili **base**, **altezza** e **area** è rappresentata in Figura 1.5, dove il

8: Ogni linguaggio ha le sue regole di validità dei nomi di variabili, quelle indicate in questo capitolo sono un sottoinsieme di quelle standard del C++, sufficiente per poter scegliere dei buoni nomi di variabile

9: Il carattere *underscore* (_) che viene spesso utilizzato per formare nomi composti da più parole, in Flowgorithm non può essere usato, cosa che invece avverrà nel linguaggio C++.

10: Il motivo del perchè nei computer esiste la distinzione tra numeri interi e numeri con la virgola è legato al fatto che la ALU (Arithmetic Logic Unit, unità aritmetico-logica) tratta in maniera separata questi due tipi numerici, con circuiti specifici per ognuno.

simbolo grafico è un rettangolo con un contorno parziale e al suo interno c'è una lista, separata da virgola, di nomi di variabili tutte delle stesso tipo, in questo caso **reale**, che vuol dire numero con la virgola. Se ci fossero state altre variabili di tipo intero sarebbe stato necessario aggiungere un nuovo blocco con l'elenco di quelle variabili, ma indicando come tipo **intero**.

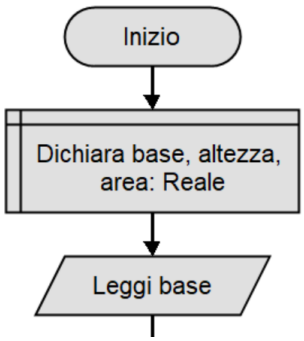


Figura 1.5.: Frammento di diagramma con la dichiarazione delle variabili

Da notare infine che la dichiarazione di ogni variabile **deve** avvenire **prima** di qualsiasi suo utilizzo.

Istruzioni di assegnamento

Nella definizione di algoritmo si parlava di "*una sequenza finita di passi elementari*", riferendosi ad essi come delle generiche *istruzioni*. Ma cos'è esattamente un'istruzione nel contesto di un algoritmo? Se l'algoritmo ad esempio fosse una ricetta di cucina, un'istruzione perfettamente valida potrebbe essere "*Aggiungi 100 gr. di zucchero*" piuttosto che "*Mescola delicatamente fino a che l'impasto non diventerà cremoso*".

Nel contesto invece di un algoritmo per computer tutte le istruzioni possono essere ridotte a *istruzioni di assegnamento*[11].

11: In alcuni testi e anche nel programma Flowgorithm vengono indicate come istruzioni di assegnazione, ma in questo testo si preferisce assegnamento.

Istruzione di assegnamento

Un'istruzione di assegnamento consiste nell'assegnare un nuovo valore a una variabile e viene rappresentata con il seguente formalismo:

```
variabile = nuovo_valore
```

dove:

- *variabile* è il nome della variabile che riceverà il nuovo valore
- *nuovo_valore* è un valore costante (un numero nei nostri esempi) oppure un'*espressione*, cioè generalmente la combinazione di valori costanti e/o altre variabili tramite, ad

esempio, gli operatori matematici di somma, differenza, ecc.

Da notare che l'istruzione di assegnamento, in cui viene utilizzato il simbolo **=**, è differente dal significato che assume lo stesso simbolo in matematica: alcuni linguaggi, per evitare problemi ai principianti, utilizzano il simbolo **:=** invece di **=**, oppure, soprattutto come rappresentazione negli algoritmi in *pseudolinguaggio*[12], il simbolo ←.

Quelli che seguono sono esempi di istruzioni di assegnamento:

Esempi di istruzioni di assegnamento

In questo esempio alla variabile **a** viene assegnato il valore 7

$$a = 7$$

In questo esempio alla variabile **b** viene assegnato il valore contenuto nella variabile **c** a cui viene sommato il valore contenuto nella variabile **d** moltiplicato per 7

$$b = c + 7 * d$$

Nel prossimo esempio, che se interpretato come un'equivalenza matematica non avrebbe senso, il valore della variabile **e** viene decrementato di 15.7 e il nuovo valore viene assegnato alla variabile **e**. Se, ad esempio, il valore della variabile **e** prima di questa istruzione fosse stato 30, dopo aver fatto questo assegnamento il nuovo valore della variabile **e** sarà 14.3.

$$e = e - 15.7$$

Attenzione: un assegnamento prevede sempre che alla sinistra ci sia una variabile (tecnicamente un *l-value*), quindi un assegnamento come il seguente **non** è valido.

$$a + b = 7$$

In un diagramma di flusso le istruzioni di assegnamento vengono rappresentate utilizzando un rettangolo contenente l'istruzione vera e propria, come si può vedere in Figura 1.6

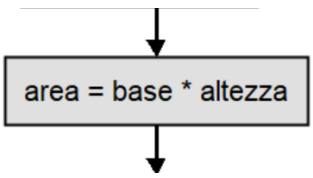


Figura 1.6.: Istruzione di assegnamento

dove la variabile **base** assume il valore ottenuto moltiplicando[13] tra loro i valori contenuti nelle variabili **base** e **altezza**.

12: Gli pseudolinguaggi sono delle versione semplificate dei linguaggi di programmazione reali, pensati per un utilizzo didattico e quindi non utilizzabili direttamente al computer

13: Generalmente in Informatica il simbolo del prodotto è l'asterisco * anzichè la ×, questo perchè altrimenti la lettera x non potrebbe essere usata.

Mettendo tutte insieme le opzioni viste finora (inizio/fine, input/output, assegnamento), l'algoritmo per il calcolo dell'area di un rettangolo, dati come input la base e l'altezza, risulta quello rappresentato dal diagramma di flusso di Figura 1.7.

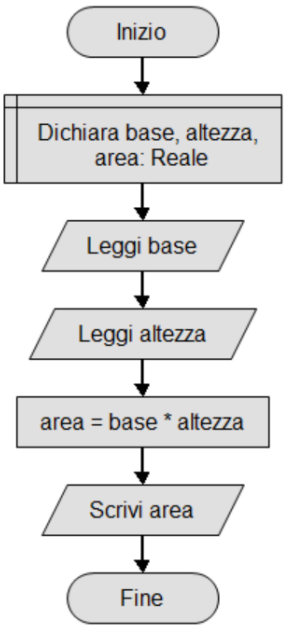


Figura 1.7.: Algoritmo per il calcolo dell'area di un rettangolo

Un'istruzione di assegnamento particolare è quella che avviene quando si vuol fare in modo che una variabile abbia un valore di partenza noto prima di essere utilizzata per la prima volta. In questo caso si parla di *inizializzazione di una variabile*, che quindi è una normale istruzione di assegnamento, solo che viene fatta **prima** del primo utilizzo della variabile. Chiaramente le variabili di input non hanno bisogno di inizializzazione, poichè il loro valore verrà fornito inizialmente dall'utente.

1.3. Istruzioni per il controllo del flusso

L'esempio di Figura 1.7 è chiaramente un esempio minimale, uno dei più semplici possibili, ma la maggior parte degli algoritmi ha una complessità di gran lunga maggiore, come si vedrà nel seguito del testo.

Dovrebbe quindi essere chiaro che quanto visto finora non è sufficiente per poter rappresentare tutti i possibili algoritmi: si vedrà però che aggiungendo soltanto altri due *costrutti*, la potenza rappresentativa dei diagrammi di flusso diventerà massima, permettendo di rappresentare tutti gli algoritmi possibili.

I costrutti che permetteranno di scrivere ogni genere di algoritmo, oltre a quello di *sequenza*, che è già stato implicitamente introdotto

nei paragrafi precedenti, sono quelli di *selezione* e di *iterazione* (o *ripetizione*).

Sebbene la dimostrazione di questo fatto vada ben oltre gli scopi del presente testo, viene enunciato di seguito il teorema di Böhm-Jacopini

Teorema di Böhm-Jacopini

Qualunque algoritmo può essere implementato in fase di programmazione (come diagramma di flusso, pseudocodice o codice sorgente) utilizzando tre sole strutture dette *strutture di controllo*: la *sequenza*, la *selezione* e l'*iterazione*.

La conseguenza di questo teorema è che qualsiasi algoritmo può essere implementato utilizzando sequenze, selezioni e iterazioni, combinandole tra loro nel modo appropriato.

Sequenza

Il costrutto di *sequenza* è già stato implicitamente presentato nel diagramma di flusso di Figura 1.7 e altro non è che la possibilità per ogni blocco di avere un blocco che lo precede e un blocco che lo segue (tranne eventualmente il primo e l'ultimo blocco). Visivamente questo semplice concetto viene reso dalla presenza delle frecce, che indicano appunto la sequenza con cui devono essere eseguite le istruzioni.

Definizione di sequenza

La *sequenza* è la struttura di controllo che viene utilizzata per rappresentare una situazione in cui le istruzioni sono eseguite una dopo l'altra.

Anche volendo modificare leggermente l'algoritmo precedente aggiungendo il calcolo del perimetro oltre a quello dell'area, si ottiene comunque un diagramma che contiene solo il costrutto di sequenza, come si può vedere in Figura 1.8: dovrebbe risultare evidente che il solo costrutto di sequenza permette di scrivere solo algoritmi molto semplici, che possono risolvere un insieme ristretto di problemi. Per aggiungere "espressività" agli algoritmi sarà quindi necessario aggiungere le due strutture di controllo che verranno presentate nei prossimi due paragrafi.

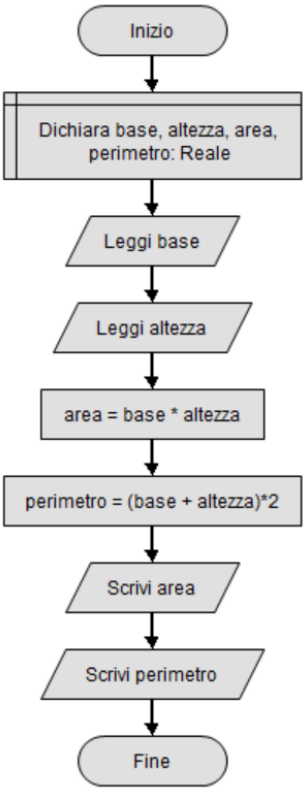


Figura 1.8.: Algoritmo con l'aggiunta del calcolo del perimetro

Selezione

Nella maggior parte dei problemi, la soluzione passa attraverso la scelta di effettuare un'azione o un'altra in base alla verità o meno di una certa condizione. Un esempio preso dalla vita reale potrebbe essere questo:

Esempio di selezione

Se oggi pomeriggio ci sarà il sole Andrea andrà al lago con gli amici, **altrimenti** rimarrà a casa a leggere un bel libro.

In questo esempio la condizione è "*oggi pomeriggio ci sarà il sole*", che chiaramente potrà essere *vera* o *falsa* a seconda delle condizioni meteorologiche. Se oggi pomeriggio splenderà il sole, allora Andrea andrà a divertirsi al lago, altrimenti lo aspetta un pomeriggio più tranquillo leggendo un libro.

Definizione di selezione

La *selezione* è la struttura di controllo che viene utilizzata per rappresentare una situazione in cui è presente una condizione, il cui valore di verità (vero o falso) determina l'esecuzione di un'istruzione (o un insieme di istruzioni) tra due possibili.

Questo tipo di struttura, che nei linguaggi di programmazione viene spesso indicata con il termine `if-else`, nei diagrammi di flusso viene rappresentata nel seguente modo:

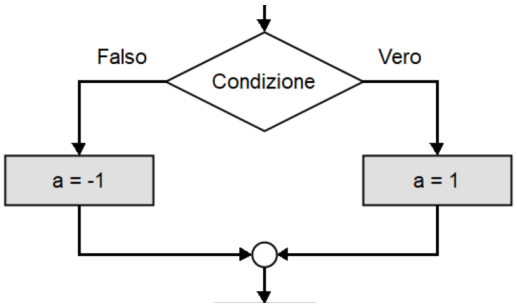


Figura 1.9.: Struttura di selezione

Aggiungendo una selezione, il flusso del programma viene suddiviso in due flussi possibili, spesso chiamati *rami della selezione*, uno per il vero e uno per il falso, che non potranno verificarsi contemporaneamente, poichè la condizione potrà essere o vera o falsa, ma non entrambe le cose. Un esempio di un problema che richiede una struttura di selezione è il seguente:

Problema con selezione

Dati il costo unitario e la quantità acquistata di un certo articolo, trovare il costo totale, considerando che se vengono acquistati più di 10 pezzi viene applicato uno sconto del 10% sul totale.

In questo problema i dati di input sono il costo unitario e la quantità acquistata, mentre l'output da mostrare a video sarà il costo totale dell'acquisto. Come si vede la presenza della selezione è determinata dal fatto che lo sconto viene applicato *solo se* vengono comprati più di 10 pezzi, che quindi è la condizione da verificare per vedere quale sarà l'azione da compiere (applicare o meno lo sconto). Il diagramma risulta quindi quello di Figura 1.10

Selezione a una sola via

Ci sono situazioni nelle quali una selezione prevede che non ci sia la parte dell'*altrimenti*, nel senso che se la condizione si verifica verrà fatta un'azione, altrimenti non succederà nulla. Nei diagrammi di flusso questo viene rappresentato semplicemente evitando di mettere un blocco nel ramo del *falso* e lasciando solo quello presente nel ramo del *vero*. Il problema visto in precedenza potrebbe anche essere risolto con l'algoritmo di Figura 1.11, dove in effetti viene evitato di inserire un'azione sul ramo del *falso*.

Va notato che, mentre nei diagrammi di flusso, si potrebbe pensare di mettere il blocco solo nel ramo del *falso* e non in quello del *vero*, ottenendo lo stesso effetto a patto di invertire la condizione, nei

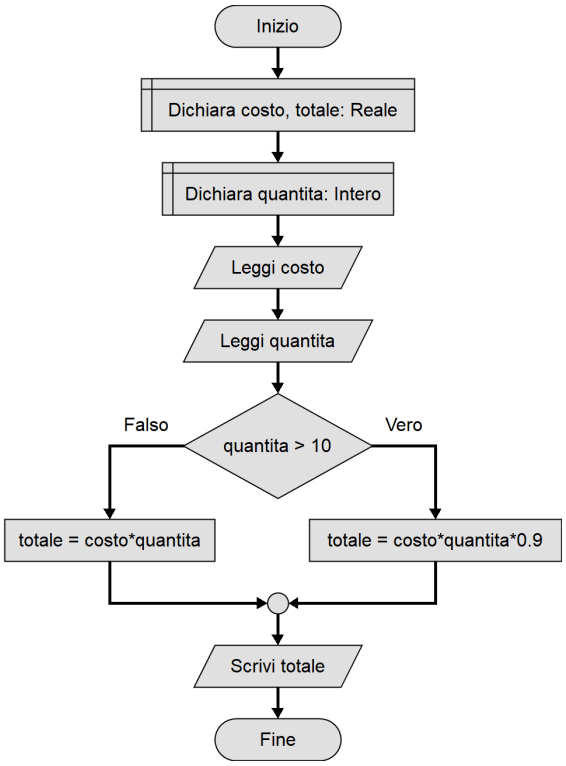


Figura 1.10.: Algoritmo per il calcolo di un eventuale sconto

linguaggi di programmazione si vedrà che, in situazioni in cui c'è una sola via, va comunque messa nel ramo del *vero*, eventualmente invertendo la condizione.

Selezioni in cascata

Problemi così semplici come quello trattato in precedenza per il calcolo dello sconto, non sono molto comuni nella realtà, che tende a essere più complessa. Ci sono ovviamente situazioni in cui una sola condizione non è sufficiente, perchè se ne vogliono controllare più di una, e una soluzione ovvia è quella di avere una *sequenza di selezioni*, una di seguito all'altra.

Si prenda ad esempio il seguente problema:

Problema con selezioni in cascata

Se il voto di una materia risulta minore o uguale a 5, allora il colore del voto sul registro apparirà rosso, se invece il voto è compreso tra 5 e 6, estremi esclusi, allora il colore del voto sul registro sarà giallo, da 6 in su invece sarà verde. Scrivere un algoritmo che, dato in input un voto, scriva in output il colore con cui sarà visualizzato a registro.

Come dovrebbe essere chiaro, questo problema presenta tre situazioni differenti, quindi non sono sufficienti due rami, quello

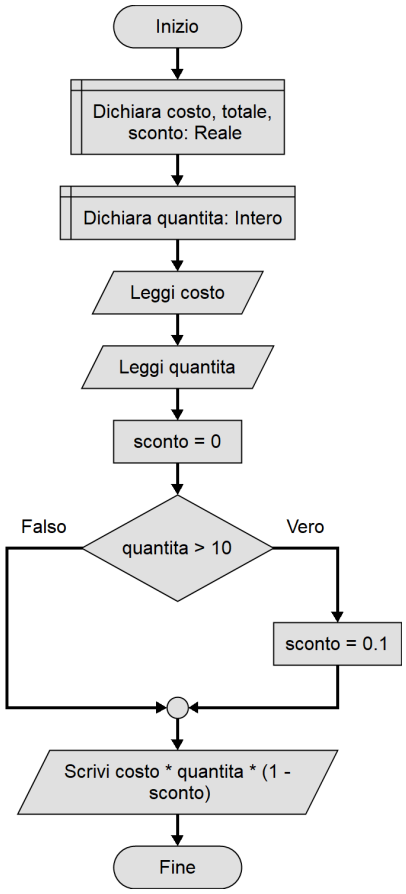


Figura 1.11.: Algoritmo per il calcolo di un eventuale sconto, versione a una via

del *vero* o del *falso*, per poterle distinguere tutte. Una soluzione possibile è quella in Figura 1.12a, dove ogni colore viene stampato o meno a seconda che la corrispondente condizione sia vera oppure falsa. In questo esempio la condizione riguardava la stessa variabile (il voto), ma in generale un algoritmo può contenere sequenze di condizioni anche applicate su variabili diverse.

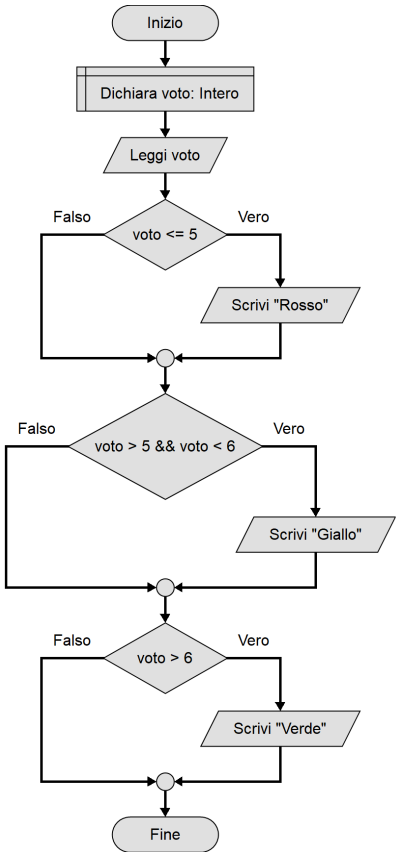
Selezioni annidate

Ci si potrebbe essere accorti che la soluzione con selezioni in cascata non garantisce *strutturalmente* che venga stampato un solo colore: per come è fatto il diagramma potrebbe anche essere possibile che non venga stampato nessun colore o addirittura tutti e tre, questo perchè i possibili percorsi dall'input all'output sono 6[14]. In questo esempio, se le condizioni sono scritte correttamente, solo un colore verrà stampato, ma la garanzia non risiede nella struttura del diagramma, quanto nell'attenzione del progettista nell'inserire le condizioni corrette.

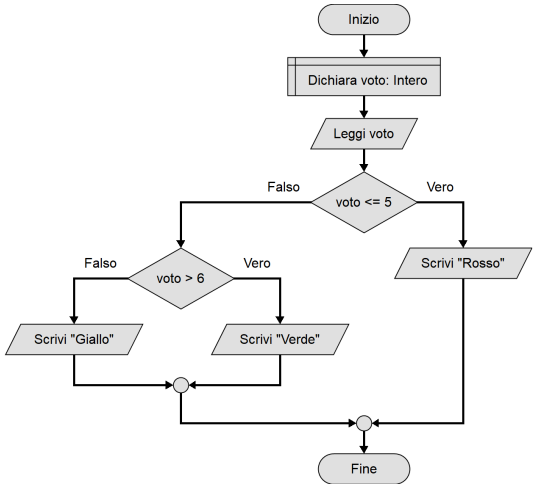
Una soluzione alternativa, in questo caso più robusta, consiste nella realizzazione di un algoritmo con *selezioni annidate*[15], come si può vedere in Figura 1.12b. Questo nuovo algoritmo risolutivo sfrutta il fatto che, se la prima condizione (voto <= 5) è falsa, allora

14: Provare a disegnare tutti i possibili percorsi diversi con colori differenti, ci si accorgerà che bisogna utilizzare 6 colori

15: In Informatica con annidamento si intende la possibilità di inserire qualcosa dentro un'altra, in questo caso una selezione all'interno di un'altra selezione.



(a) Soluzione con selezioni in cascata



(b) Soluzione con selezioni annidate

Figura 1.12.: Soluzione del problema dei voti con selezioni in cascata e selezioni annidate

il voto è per forza maggiore di 5 e a questo punto rimangono solo gli intervalli tra 5 e 6 e quello maggiore di 6, che possono essere distinti utilizzando un'ulteriore condizione. Stavolta la struttura del diagramma implica che *uno e un solo* colore potrà essere stampato ad ogni esecuzione dell'algoritmo, indipendentemente dal fatto che le condizioni siano corrette o meno. Può essere questo un buon motivo per usare selezioni annidate piuttosto che selezioni in cascata? Purtroppo o fortunatamente, come si vedrà spesso nel seguito, non esistono regole "automatiche" che permettono al progettista di algoritmi di fare delle scelte senza dover pensare: ogni scelta, anche la più semplice, va valutata nel contesto del problema che si vuole risolvere e solo in questo modo si possono fare le scelte corrette.

Iterazione

L'ultima delle strutture di controllo che verrà analizzata ora, l'*iterazione* o *ripetizione*, è anche quella più "potente", che permette sostanzialmente di ripetere una serie di azioni.

Definizione di iterazione

L'*iterazione* è la struttura di controllo che viene utilizzata per rappresentare una situazione in cui è presente un'istruzione (o un insieme di istruzioni) che viene ripetuta finchè una certa condizione rimane vera.

Per fare un semplice esempio, si supponga di avere un algoritmo che permetta di rendere la propria tazza di cioccolata dolce al punto giusto. L'algoritmo potrebbe essere scritto nei due modi mostrati in Tabella 1.1: nella prima colonna è stato scritto sotto forma di una sequenza di selezioni (se non è dolce abbastanza), ognuna seguita da un'azione (aggiungi un cucchiaio di zucchero), mentre la seconda colonna contiene la versione iterativa, rappresentata nella lingua italiana dalla congiunzione *finchè*.

Tabella 1.1.: Confronto tra sequenza e iterazione

Sequenza	Iterazione
Se non è dolce abbastanza Aggiungi un cucchiaio di zucchero Se non è dolce abbastanza Aggiungi un cucchiaio di zucchero Se non è dolce abbastanza ...	Finchè non è dolce abbastanza, aggiungi un cucchiaio di zucchero

Si può vedere come entrambi gli "algoritmi" dicano la stessa cosa, però quello con la sequenza non è chiaro come possa essere trasformato in un algoritmo reale, poichè non è possibile sapere quante volte sarà necessario controllare la dolcezza del cioccolato e quindi aggiungere un cucchiaio di zucchero, tanto è vero che è stato necessario aggiungere i puntini di sospensione per indicare la mancanza di quest'informazione.

Viceversa la versione iterativa risolve il problema in maniera elegante, risultando altrettanto corretta sia che i cucchiaini da versare siano due piuttosto che mille. Si vedrà ora come rappresentare questa struttura di controllo nei diagrammi di flusso.

Iterazione con cicli indeterminati

L'idea fondamentale che sta alla base dell'iterazione è, come si è detto, la possibilità di ripetere delle azioni fino a quando una certa condizione rimane vera. Nei diagrammi di flusso l'iterazione viene rappresentata nel seguente modo[16]:

La figura rappresenta, sempre tramite l'utilizzo delle frecce, che il flusso può seguire due percorsi:

16: Si noti che Flowgorithm utilizza la parola *mentre* anzichè *finchè*, come indicato in questo testo. Si vedrà in seguito che la parola inglese utilizzata nei linguaggi di programmazione sarà `while`, possono tutte essere considerate sinonimi.

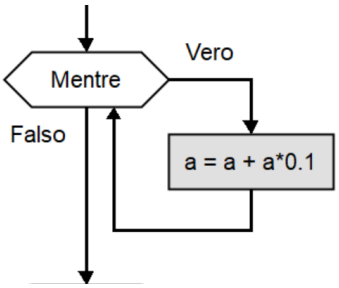


Figura 1.13.: Struttura di iterazione con cicli indeterminati

- se la condizione all'interno del blocco con il *mentre* risulta vera, allora verrà eseguita l'azione o le azioni all'interno di quel ramo, che nella figura d'esempio è l'assegnamento `a = a + a*0.1`
- quando poi la condizione diventerà falsa, allora il flusso d'esecuzione procederà oltre questo blocco, svolgendo le istruzioni che si troveranno dopo la freccia in basso.

Si vedrà adesso un problema che permetterà di vedere come applicare praticamente questo concetto.

Problema con iterazione determinata

Scrivere un algoritmo che, dato in input un numero intero positivo N, trovi il più grande dei quadrati perfetti minore del numero dato N. Si ricorda che un quadrato perfetto è un numero intero che può essere ottenuto come prodotto di un certo altro numero intero per se stesso.

Un modo per risolvere questo problema è di seguire questo ragionamento: non avendo idea di quale possa essere la soluzione richiesta, si può chiedere al computer di calcolare tutti i quadrati perfetti a partire dal numero 1 in poi, e, ad ogni nuovo quadrato, verificare se sia o meno minore di N. *Finchè* risulta minore di N può essere un candidato alla soluzione, ma potrebbe essercene uno più grande, e quindi si procede a provare con il quadrato perfetto successivo. A un certo punto si troverà per forza un quadrato perfetto maggiore o uguale a N, il che significherà che il quadrato precedente era la soluzione cercata, essendo sicuramente il più grande quadrato tra tutti quelli calcolati che rispetta la condizione di essere minore di N. Questo ragionamento trasformato in un diagramma di viene rappresentato in Figura 1.14.

Questo tipo di iterazione viene detta *iterazione con cicli*[17] *indeterminati*, o semplicemente *iterazione indeterminata*, poichè il numero di cicli che verranno compiuti all'esecuzione dell'algoritmo non è noto a priori, dipendendo dall'input che verrà dato dall'utente, in questo caso dal valore di N.

17: Per ciclo si intende il singolo passaggio nel ramo del vero.

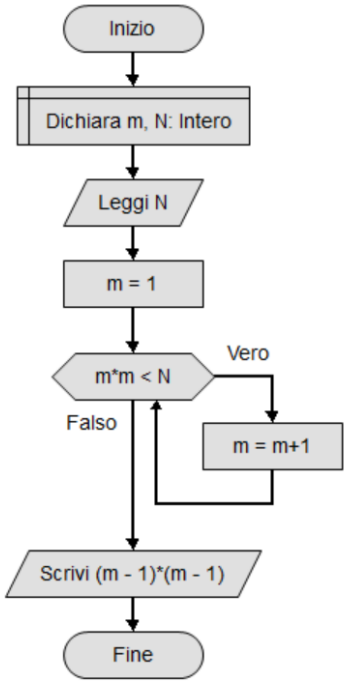


Figura 1.14.: Esempio di algoritmo con iterazione con cicli indeterminati

Iterazione con cicli determinati

La modalità presentata nel paragrafo precedente è la struttura più generale per poter rappresentare algoritmi che al loro interno contengono delle iterazioni. In pratica però viene spesso introdotto anche un ulteriore costrutto che permette di rappresentare delle iterazioni, questa volta con *cicli determinati*. Per cicli determinati si intende il fatto che, in alcuni algoritmi, il numero di cicli è già conosciuto a priori, quindi diventa comodo avere una struttura che rappresenti questa informazione esplicitamente.

Non tutti gli strumenti per rappresentare diagrammi di flusso permettono il disegno di questa struttura di controllo, ma, poichè Flowgorithm lo permette e anche in C++ è presente, verrà utilizzata nelle situazioni opportune. La rappresentazione che ne da Flowgorithm può essere vista nella Figura 1.15. L'esagono che contiene l'istruzione `i = 1 to 10` indica che l'iterazione verrà ripetuta esattamente 10 volte e **i** viene detta anche *contatore* dei cicli, poichè ad ogni ciclo viene automaticamente aumentata di un'unità, partendo da 1 e arrivando fino a 10 compreso [18].

La motivazione alla base dell'introduzione di questo costrutto, spesso chiamato **`for`** nei linguaggi di programmazione, è che molti problemi prevedono delle soluzioni nelle quali viene fatto un numero di iterazioni noto, o perchè direttamente inserito dall'utente come dato o perchè risultato di una qualche elaborazione. Un esempio è il seguente:

18: Nei cicli che utilizzano il *for* è prassi comune utilizzare come variabile contatore la lettera **i**

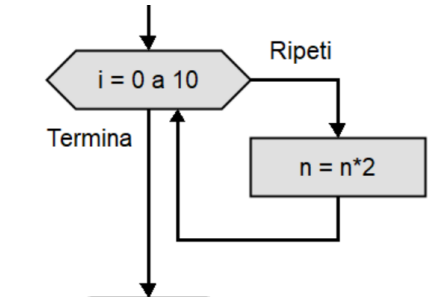


Figura 1.15.: Iterazione con cicli determinati

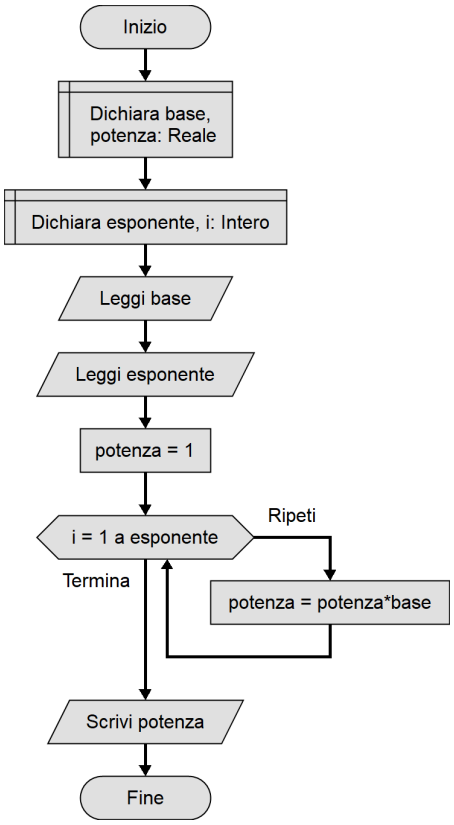


Figura 1.16.: Esempio di algoritmo con iterazione con cicli determinati

Problema con iterazione indeterminata

Dato un numero razionale **b** che rappresenta la base di un elevamento a potenza e un numero intero positivo **e** che ne rappresenta l'esponente, calcolare la potenza b^e.

Ricorrendo alla definizione di potenza con esponente intero, cioè alla moltiplicazione della base per se stessa il numero di volte indicato dall'esponente, è semplice disegnare il diagramma di flusso corrispondente, che può essere visto in Figura 1.16.

Lo stesso algoritmo si sarebbe anche potuto scrivere utilizzando un ciclo indeterminato, avvalendosi di un *contatore*, cioè una variabile da incrementare ad ogni ciclo, e mettendo come condizione un controllo per verificare che il contatore fosse minore di 10. La soluzione che utilizza però i cicli determinati è più compatta e viene spesso preferita quando ci si trova in questi tipi di problemi.

Come scegliere se utilizzare un `while` (cicli indeterminati) o un `for` (cicli determinati)? Non esiste una regola certa, però vanno tenute a mente queste indicazioni:

- se la condizione di controllo del ciclo non è basata sul conteggio del numero dei cicli, ma sul verificarsi di un certo evento che non si sa quando succederà, allora la scelta più ovvia è quella di utilizzare un `while`
- se la soluzione del problema prevede già, implicitamente o esplicitamente, che si sappia il numero di cicli che andranno svolti, allora la scelta sarà quella di utilizzare un `for`
- con un po' di esperienza risulta automatico, per quasi tutti i problemi, individuare quando utilizzare una struttura piuttosto che l'altra: vale la pena comunque notare che, mentre un problema che tipicamente verrebbe risolto con un `for` può essere risolto in maniera simile utilizzando un `while`, il viceversa, pur essendo in generale possibile, solitamente prevede una scrittura piuttosto contorta del codice e in Flowgorithm non si può proprio.

1.4. Come progettare un algoritmo

Come detto, le tre strutture di controllo viste finora, *sequenza*, *selezione* e *iterazione*, permettono di risolvere qualsiasi tipo di problema: risulta comunque non banale imparare a combinarle opportunamente per scrivere un qualsiasi algoritmo risolutivo.

Quello che segue è un elenco di consigli che, se seguiti con cura, possono aiutare chi è alle prime armi nella progettazione di algoritmi:

- può sembrare ovvio, ma il passaggio più importante è quello di leggere con attenzione il testo del problema per individuare correttamente la richiesta fatta. Se nella realtà il problema viene definito generalmente attraverso un colloquio tra il programmatore e il committente dell'applicazione, durante l'apprendimento i problemi sono dei testi, volutamente semplificati, che hanno lo scopo specifico di condurre l'allievo verso certe idee risolutive. Risulta quindi chiaro che se il testo non viene compreso nella sua interezza o addirittura viene malamente frainteso, non ci sarà nessuna possibilità di arrivare a una soluzione corretta
- nei testi dei problemi didattici vengono spesso fatte delle *ipotesi semplificative*, cioè vengono definite delle situazioni che generalmente semplificano la soluzione del problema, evitando di dover gestire una serie di casistiche che, sebbene

nella pratica potrebbero presentarsi, vengono ritenute scarsamente rilevanti ai fini dello scopo didattico del problema. Esempi tipici sono, ad esempio, di assumere come sempre corretto l'input dell'utente, di considerare solo i numeri positivi, ecc.

- individuare quali sono le variabili di input e quelle di output: come detto all'inizio del capitolo, qualsiasi algoritmo può essere visto come il "passaggio" da una serie di variabili in ingresso a una serie di variabili in uscita e quindi risulta fondamentale individuarle prima di iniziare a pensare all'algoritmo risolutivo
- solo dopo aver fatto quanto descritto ai passi precedenti si può iniziare a progettare l'algoritmo vero e proprio, ma a questo punto potrebbe non essere semplice capire da che parte iniziare. Un suggerimento pratico è quello di non pensare ai diagrammi di flusso, ma focalizzarsi su come si risolverebbe il problema avendo a disposizione solo carta, penna ed eventualmente una calcolatrice[19]. Nel tentativo di arrivare all'ideazione dell'algoritmo inoltre può essere utile partire da un esempio con dei valori definiti, possibilmente in modo da essere significativi per il problema in oggetto. Non tutti i problemi vengono risolti allo stesso modo dagli umani e dai computer, ma spesso i problemi semplici non sono altro che la traduzione di un procedimento che potrebbe fare anche una persona, con la differenza che il computer lo svolgerà molto più velocemente e senza commettere errori. Vale solo la pena di ricordare che se l'algoritmo è sbagliato, il computer lo eseguirà comunque molto più velocemente e senza commettere errori, ma il risultato sarà comunque sbagliato.

19: L'utilizzo della calcolatrice serve perchè spesso i problemi iniziali sono di carattere matematico, poichè più semplici da descrivere e di interpretazione non ambigua.

Si vedranno adesso dei problemi di esempio per vedere in pratica come arrivare a degli algoritmi risolutivi partendo da dei testi.

Esempio 1: maggiore di tre numeri

Testo: dati[20] tre numeri A, B, C, stampare il maggiore dei tre.

20: Nei testi didattici quando viene usata l'espressione "dati ..." si intende che quei valori sono forniti in input dall'utente.

Considerazioni: come si può vedere il problema è molto semplice, ma risulta interessante perchè illustra un aspetto importante: le operazioni elementari che un computer è in grado di svolgere non sempre corrispondono a quelle che potrebbe fare un umano. In questo caso, dati ad esempio i tre numeri 5, 9 e 4, è immediato per l'umano vedere che il maggiore è il 9, si potrebbe dire che questa soluzione viene trovata a "colpo d'occhio". Un computer non ha un'istruzione equivalente per verificare istantaneamente il maggiore tra tre numeri, ma, come presentato in Appendice A,

è solo in grado di confrontare due numeri tra loro e dire quale dei due è il maggiore o, equivalentemente, il minore. Con questa limitazione l'algoritmo dovrà trovare la soluzione confrontando i numeri tra loro a due a due e quindi non sarà immediatamente possibile stabilire quale sia il maggiore. Invece si seguirà questa linea di ragionamento:

- dati i tre numeri **A**, **B** e **C**, se ne prendano due, ad esempio **A** e **B**, e si trovi il più grande, che verrà memorizzato in una variabile *d'appoggio*[21]. Questo controllo permette di escludere uno dei due numeri, il minore dei due, poichè sicuramente non può essere quello il maggiore.
- si prenda il numero che è stato trovato essere il maggiore al passo precedente e lo si confronti con **C**, per vedere chi dei due è il più grande. Il maggiore tra questi due è anche il maggiore assoluto tra **A**, **B** e **C**.

21: Spesso ci si riferisce a delle variabili che non sono né quelle di input, né le soluzioni del problema, come variabili d'appoggio, poichè servono per *appoggiare* temporaneamente dei valori che saranno poi utilizzati in qualche altra parte dell'algoritmo.

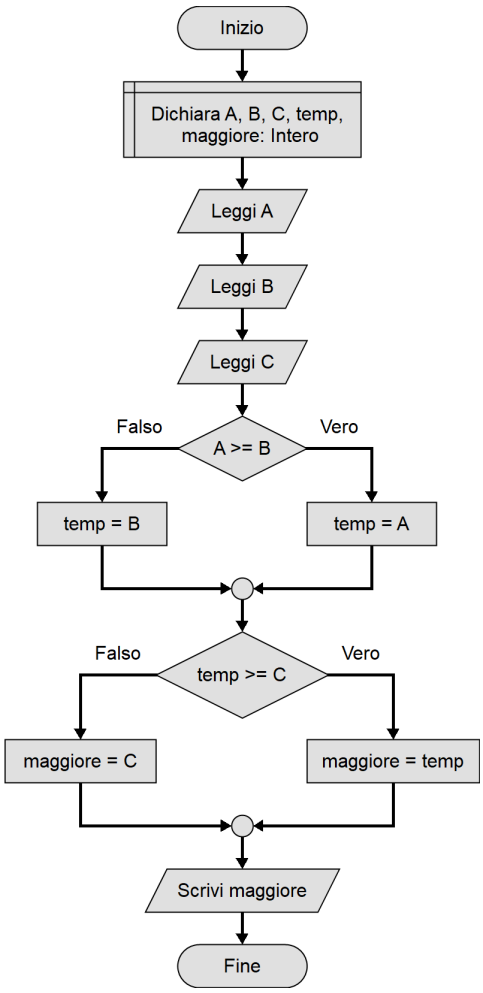


Figura 1.17.: Esempio 1: maggiore di tre numeri

Questa idea viene rappresentata nel diagramma di flusso di Figura 1.17.

Approfondimenti

- Riflettere sul perchè è stato usato l'operatore **>=** anzichè semplicemente **>** e a cosa succederebbe in caso contrario.
- Pensare a una versione alternativa che porti alla stessa soluzione. *Suggerimento*: si pensi a come sfruttare l'operatore logico **and** (&&) per combinare opportunamente più condizioni.

Esempio 2: calcolo del fattoriale di un numero

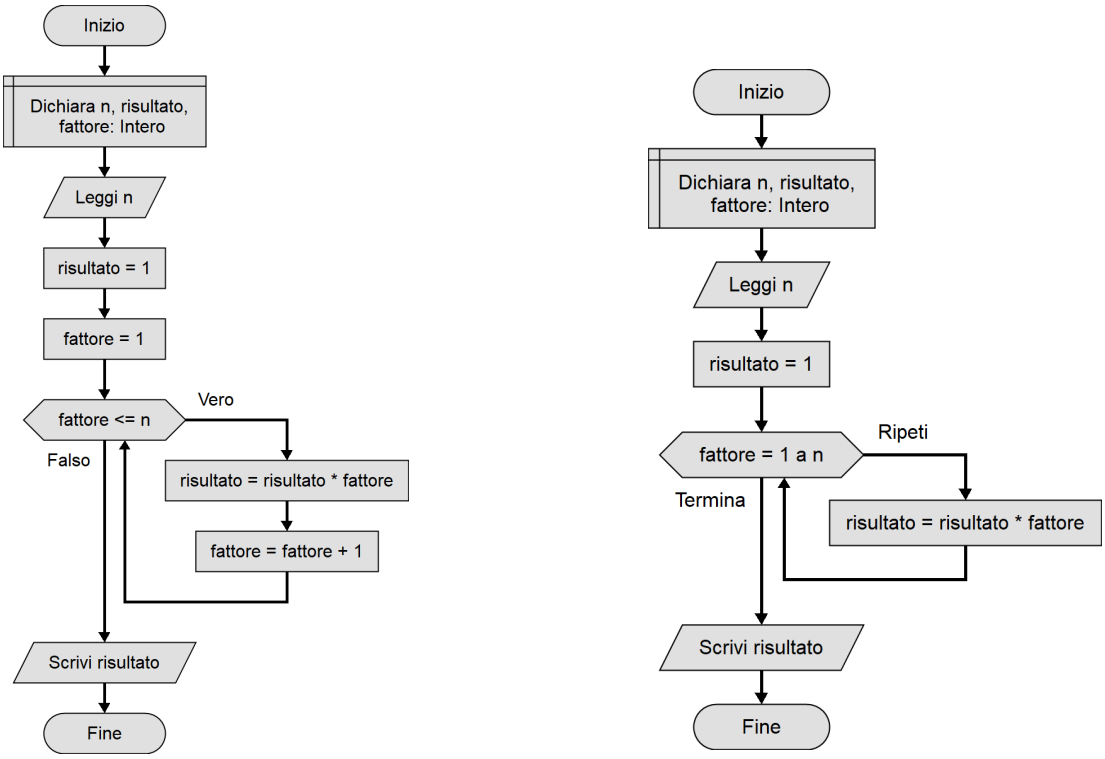
Testo: dato un numero intero positivo **n**, calcolare il suo fattoriale.

Considerazioni: come prima cosa si può vedere che in questo testo viene richiesto il calcolo di una funzione matematica che potrebbe non essere nota al lettore, il *fattoriale*. Siccome il prerequisito fondamentale per scrivere un algoritmo risolutivo è, come si è detto, la comprensione corretta e completa del testo del problema, bisognerà prima scoprire il significato della parola *fattoriale*. Con una semplice ricerca su Internet o su un libro di matematica che tratta ad esempio di combinatoria, si può trovare la seguente definizione:

> Si definisce *fattoriale* di un numero naturale **n**, indicato con **n!**, il prodotto dei numeri interi positivi minori o uguali a tale numero. Ad esempio il fattoriale di 5, indicato con 5!, vale
>
> $$1 \times 2 \times 3 \times 4 \times 5 = 120$$

Anche in questo caso "calcola il fattoriale di un numero" non è un'istruzione elementare che può essere svolta direttamente dal computer e quindi è necessario trovare un algoritmo che calcoli il fattoriale come una serie di moltiplicazioni, che invece sono operazioni elementari che il computer è in grado di svolgere. Il problema è che il numero di operazioni da svolgere non è fisso ma dipende dal valore di **n**, che verrà fornito dall'utente e che quindi risulta ignoto al tempo di scrittura dell'algoritmo, impedendo di sapere quante saranno le moltiplicazioni da fare. In queste situazioni deve diventare naturale accorgersi che la soluzione passerà attraverso l'utilizzo di un'*iterazione*, poichè abbiamo un'operazione elementare, che in questo caso è la moltiplicazione, che verrà ripetuta un certo numero di volte. L'algoritmo potrà quindi essere progettato in questo modo:

1. inizializzare al valore 1 una variabile **risultato**, che farà da *accumulatore*
2. inizializzare al valore 1 un'ulteriore variabile **fattore**, che conterrà il valore da moltiplicare ad ogni singolo passaggio
3. moltiplicare il **fattore** per il **risultato** e assegnare il prodotto ottenuto a **risultato**



(a) Soluzione con *while*

(b) Soluzione con *for*

Figura 1.18.: Soluzione del problema del calcolo del fattoriale

4. aumentare di un'unità il valore del **fattore** e:
   - se il **fattore** è minore o uguale a **n** continuare ripartendo dal punto 3
   - altrimenti il programma termina e viene stampato il valore contenuto in **risultato**, che sarà il fattoriale di **n**.

Questo algoritmo può essere rappresentato sia attraverso l'utilizzo di un `while` che utilizzando un `for`, come si può vedere in Figura 1.18, anche se tipicamente verrebbe fatto utilizzando il secondo costrutto.

L'idea di una variabile *accumulatore* è molto potente e si ritrova in moltissimi algoritmi elementari, poichè permette di "accumulare" i risultati di operazioni ripetitive un passo alla volta, per arrivare ad avere al suo interno il risultato desiderato.

Un caso particolare di accumulatore è il *contatore*, cioè una variabile che ad ogni passo viene incrementata di un'unità. Anche in questo caso si ha un processo di accumulazione, poichè ad ogni passo il contatore aumenterà di uno, come si può vedere per la variabile **fattore** degli algoritmi mostrati. Il nome deriva dal fatto che queste genere di variabili hanno lo scopo di *contare* quante volte si ripete una certa operazione e permettere l'uscita da un'iterazione al raggiungimento di un certo valore. Inoltre, come visto in queto

esempio, il valore di un contatore può anche essere usato per svolgere qualche tipo di operazione ad ogni ciclo dell'iterazione.

Approfondimenti

- Negli algoritmi mostrati i valori di **risultato** e **fattore** vengono entrambi inizializzati a 1: pensare a cosa sarebbe successo se fossero stati inizializzati a valori diversi, in particolare a 0 oppure a 2 e discuterne le conseguenze per ogni possibile alternativa.
- Nella versione che utilizza il ciclo `for`, viene usato direttamente **fattore** sia come contatore del ciclo, sia come valore per la moltiplicazione con risultato. Discutere se si sarebbe potuto utilizzare la variabile **i** come visto nell'iterazione con cicli determinati, cosa sarebbe cambiato e perchè utilizzare una forma piuttosto che l'altra.

1.5. Considerazioni finali

In questo capitolo si è visto cos'è un *algoritmo*, quali sono i blocchi elementari con cui costruirlo attraverso la notazione dei *diagrammi di flusso* e si sono visti degli esempi per capire come passare dalla descrizione di un problema alla stesura di un algoritmo per risolverlo.

Come si avrà modo di notare nel seguito del testo, la difficoltà principale risiede proprio nel passaggio dalla descrizione del problema all'ideazione dell'algoritmo risolutivo: il modo più rapido per imparare a fare questo passaggio è quello di vedere molti problemi e provare a risolverli, eventualmente riprovando più volte la soluzione del problema anche a distanza di tempo, per essere sicuri di riuscire a risolverlo autonomamente.

Stesso problema, algoritmi risolutivi differenti

All'inizio del capitolo era stato proposto un esempio di problema e tre diversi algoritmi per risolverlo (si veda pagina 4). Per mettere in evidenza che la soluzione di un problema **può** passare attraverso idee diverse, ma tutte corrette, si farà una breve analisi dei tre algoritmi proposti con la realizzazione dei rispettivi diagrammi di flusso.

Algoritmo 1

La soluzione proposta era la seguente

Si prenda ogni numero da 1 fino a N e si verifichi se è o meno una potenza di due: se lo è, si aumenti di un'unità il conteggio delle potenze del 2, altrimenti non si faccia niente.

Questo è un esempio di approcio a **forza bruta**, dove cioè si utilizza la velocità di calcolo del computer per trovare la soluzione per tentativi. Anche se all'apparenza questa soluzione può sembrare semplice da implementare, la condizione *si verifichi se è o meno una potenza del 2* è tutt'altro che banale, in quanto non fa parte delle istruzioni elementari che un calcolatore è in grado di eseguire direttamente.

Nasce quindi la necessità di ridurre quel controllo a un "*sottoalgoritmo*" che il computer sappia svolgere utilizzando solo operazioni elementari: in questo caso si può notare che le potenze del 2 hanno la proprietà che possono essere divise continuamente per 2 senza mai produrre un resto[22], fino a quando non si arriva a 1.

22: Questo perchè un numero potenza del 2 è della forma $2 \times 2 \times 2 \times \ldots \times 2$

L'implementazione può essere vista in Figura 1.19, dove si può notare che la parte più complessa è proprio il sottoalgoritmo di verifica della potenza del 2: infatti è necessario utilizzare un `while` che divida continuamente per 2 un numero *candidato* e, per ogni ciclo, verifichi che il resto non diventi diverso da zero, nel qual caso dovrà modificare la variabile *flag*[23] **potenzaDelDue** per indicare che l'ipotesi che il candidato fosse una potenza del 2 è falsa. Se, dopo aver effettuato tutte le divisioni, non si avrà mai un resto diverso da zero, allora il candidato è una potenza del due e quindi verrà incrementato il contatore delle potenze del due.

23: Una variabile viene indicata con il termine di variabile *flag* quando il suo scopo è di indicare se una situazione si è verificata o meno, quindi spesso è di tipo *booleano*

Questo algoritmo risolutivo, pur essendo corretto, presenta due problemi:

1. il numero di operazioni necessarie si può intuire che è alto, poichè devono essere controllati tutti gli N numeri interi che sono possibili candidati e ognuno di questi controlli è a sua volta lungo, perchè prevede una serie di divisioni. In generale questo problema, la scarsa efficienza, non verrà considerato importante ai fini della progettazione di un algoritmo dato lo scopo didattico dei problemi affrontati, ma è chiaro che in nella realizzazione di programmi reali potrebbe assumere un'importanza notevole, soprattutto in alcuni contesti
2. l'enunciato risolutivo sembrava piuttosto innocuo, ma si è rivelato invece complesso poichè una condizione che veniva esposta con una frase in linguaggio naturale, non era purtroppo realizzabile tramite un'istruzione elementare del calcolatore. Questo a costretto quindi a implementare un sottoalgoritmo di una complessità maggiore del problema che si voleva risolvere.

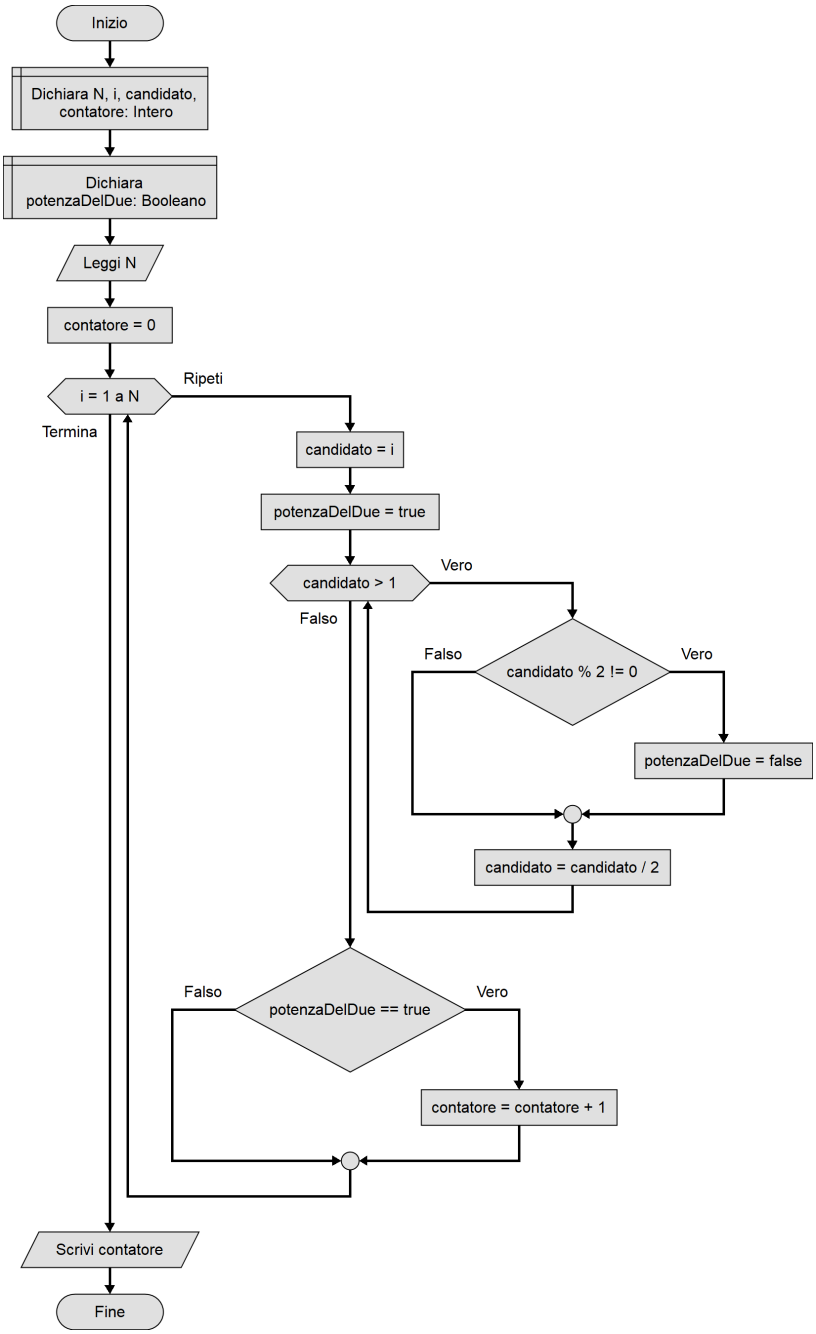


Figura 1.19.: Algoritmo 1

Algoritmo 2

In questo caso la soluzione proposta era la seguente

> Si parta dal numero 1 e lo si raddoppi aumentando contemporaneamente il conteggio delle potenze del 2. Quando questo processo porta alla produzione di un numero maggiore di N ci si fermi e il valore del conteggio ottenuto è la soluzione.

Questa soluzione, pur non essendo magari così immediata da pensare come quella presentata nell'algoritmo precedente, risolve

i due problemi presenti nell'algoritmo 1: è decisamente più efficiente e le operazioni richieste sono tutte elementari, quindi non necessitano di essere scomposte in ulteriori operazioni.

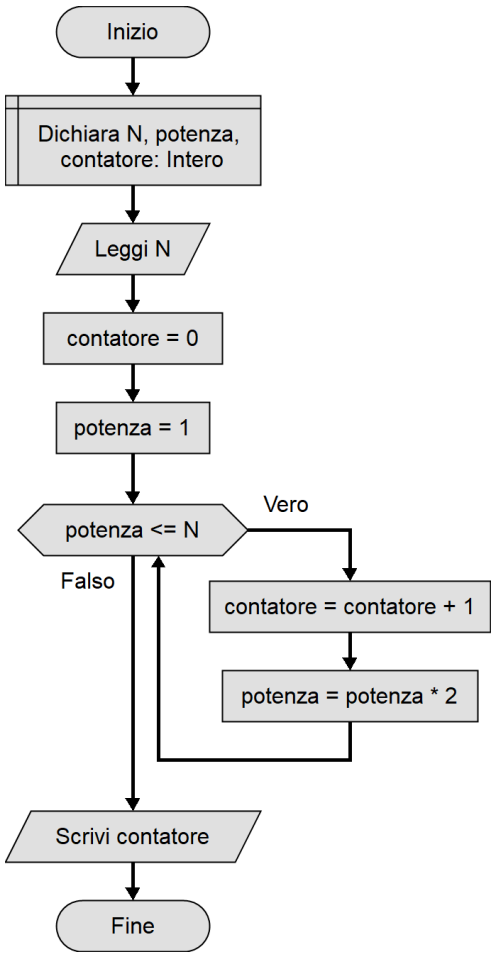


Figura 1.20.: Algoritmo 2

Si può notare anche visivamente dalla Figura 1.20 che l'algoritmo è decisamente più semplice, più elegante e più immediato da implementare, perchè rappresenta esattamente il testo della soluzione proposta.

Si può quindi osservare che la progettazione di un algoritmo influisce direttamente sulla semplicità o meno della sua implementazione, perciò soffermarsi qualche minuto in più pensando all'ideazione di un algoritmo può essere più vantaggioso che implementare direttamente la prima idea che viene in mente.

Algoritmo 3

Infine l'ultima soluzione era la seguente

> Si applichi la formula matematica
>
> $$soluzione = \lfloor log_2 N \rfloor + 1$$

Il vantaggio di questa soluzione è la semplicità implementativa, che può essere vista in Figura 1.21, dove si può notare che non appaiono né selezioni né iterazioni, ma si tratta di una sequenza composta da solo tre istruzioni. D'altra parte la conoscenza richiesta per utilizzare la formula esposta richiede delle competenze matematiche che non è detto siano nel bagaglio di ogni programmatore. Inoltre viene richiesto di saper utilizzare delle *funzioni* e di saperle combinare tra di loro, poichè in questo strumento non esiste la possibilità di calcolare direttamente il logaritmo di 2, ma è necessario ottenerlo attraverso la formula del cambiamento di base[24].

24: Se questa parte di matematica risultasse incomprensibile, non è un problema, serve a rimarcare meglio il fatto che *più corto* non vuol dire necessariamente *più facile*

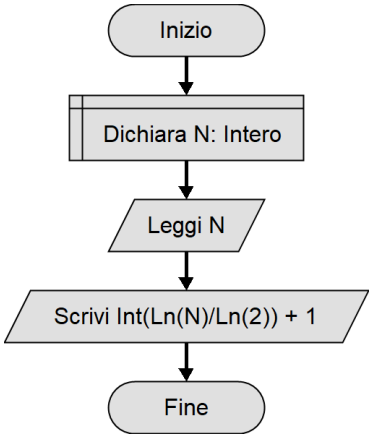


Figura 1.21.: Algoritmo 3

Nel resto del testo si preferiranno quindi le soluzioni come l'algoritmo 1 e l'algoritmo 2, dove le operazioni utilizzate sono elementari e non richiedono particolari competenze che non siano il ragionamento logico e le basi della matematica.

1.6. Esercizi

Esercizi introduttivi

1. Scrivere almeno 3 esempi di trattamento dell'informazione, cercando di descriverli a un livello di dettaglio sufficiente che permetta al lettore di farsi un'idea chiara di cosa si sta parlando.
2. Scrivere almeno 3 problemi della vita quotidiano e dare una descrizione testuale dell'algoritmo per risolverli.
3. Scrivere almeno 3 esempi di algoritmi che risolvono un problema specifico e generalizzarli come visto nell'esempio. del calcolo dell'area del rettangolo.
4. Scrivere i nomi delle variabili di input e di output per 3 algoritmi a propria scelta, differenti da quelli proposti nell'esercizio precedente, facendo in modo che siano significativi e rispettono le regole esposte in questo capitolo.
5. Scrivere in italiano almeno 3 esempi di selezione presi dalla propria esperienza quotidiana.
6. Scrivere in italiano almeno 3 esempi di iterazione presi dalla propria esperienza quotidiana.
7. Implementare i diagrammi di flusso degli esempi mostrati nei paragrafi di questo capitolo, leggendo il testo del problema e provando a implementare autonomamente l'algoritmo risolutivo. Successivamente confrontare la soluzione sviluppata con quella proposta nel testo, correggendo eventuali errori e riflettendo sulle eventuali differenze riscontrate.

Esercizi che si risolvono usando solo sequenze e selezioni

8. Dato il valore corrente del cambio bitcoin-euro, che dovrà essere recuperato da Internet, scrivere un algoritmo che, data in input una certa quantità di euro, fornisca in output il numero di bitcoin che possono essere acquistati.
9. Si supponga che per spedire i pacchi postali siano presenti queste tre tariffe

Nome	Costo
1) Normale	3 euro al Kg + 3 euro di costo fisso
2) Celere 1	4 euro al Kg + 7 euro di costo fisso
3) Celere 2	6 euro al Kg + 13 euro di costo fisso

   Data la tariffa scelta (individuata dal proprio indice 1, 2 o 3) e il peso del pacco, stabilire il costo di spedizione.
10. Dato il costo dell'abbonamento a 10 ingressi in piscina, il costo del biglietto singolo e il numero di ingressi che si vogliono fare, stabilire se convenga comprare l'abbonamento oppure i biglietti singoli, nell'ipotesi che i biglietti che si vogliono acquistare siano <= 10.
    Ripetere l'esercizio stavolta senza il limite dei 10 biglietti: stabilire il numero di abbonamenti e biglietti singoli che conviene comprare per spendere il meno possibile. Si può supporre che il costo dell'abbonamento sia sempre inferiore al costo dell'acquisto di 10 biglietti singoli.
11. Dato un numero intero positivo N, stampare "Fizz" se il numero è multiplo di 3, stampare "Buzz" se è multiplo di 5 e stampare entrambe le scritte se è sia multiplo di 3 che di 5.
12. Dato un voto numerico compreso tra 1 e 10, stampare se il voto è sufficiente oppure no, considerando 6 la soglia per la sufficienza.

Esercizi che si risolvono usando anche il costrutto di iterazione

13. Utilizzando la formula che definisce lo spazio percorso da un corpo in caduta libera come

$$s(t) = \frac{1}{2}gt^2$$

e supponendo che lo stesso sia lanciato da un'altezza h, mostrare dove si trova il grave rispetto a terra a intervalli di un decimo di secondo, fermandosi quando raggiunge il suolo.
14. Data una certa quantità di Cesio-137 espressa in grammi, stabilire dopo quanto tempo la quantità diventerà meno di un milligrammo, considerando che il tempo di decadimento è di circa 30 anni (il tempo di decadimento è il tempo che una certa massa di materiale impiega a ridursi del 50%).
15. Dato un numero intero positivo, determinare da quante cifre è composto.
16. Dati una serie di interi positivi in input, con il numero 0 che termina l'immissione, calcolare qual è la loro somma e quanti numeri sono stati immessi, supponendo che venga immesso almeno un numero.
17. Dato un numero razionale b e un numero intero e, calcolare il valore di b^e. **Nota**: essendo E intero, può essere sia positivo che negativo.
18. Progettare un algoritmo che verifica se N numeri presi in input sono forniti in ordine crescente, con N fornito dall'utente.
19. Dato un numero intero positivo n, calcolare il suo fattoriale $n!$. In matematica, si definisce **fattoriale** di un numero naturale n, indicato con $n!$, il prodotto dei numeri interi positivi minori o uguali a tale numero.

$$n! := \prod_{k=1}^{n} k = 1 \times 2 \times 3 \times \cdots (n-1) \times n$$

Inoltre si definisce $0! := 1$. Esempio: 5!, che va letto come 5 fattoriale, è uguale a 120, poichè è $1 \times 2 \times 3 \times 4 \times 5$.
20. Dato un numero n intero positivo, calcolare l'n-esimo numero di Fibonacci. In matematica, la successione di Fibonacci è una successione di numeri interi in cui ciascun numero è la somma dei due precedenti, eccetto i primi due che sono, per definizione, 0 e 1. L'n-esimo numero di Fibonacci, indicato con F_n, è quindi l'n-esimo numero di detta successione, con $F_0 = 0$ e $F_1 = 1$. I primi elementi di questa successione sono quindi 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
21. Dopo aver fatto inserire un numero intero positivo N all'utente, far inserire N numeri e stampare la loro somma a gruppi di 4. Se N non fosse divisibile esattamente per 4, l'ultimo gruppo sarà composto da meno di 4 elementi e il programma ne stamperà la somma e il numero di elementi contenuti.
22. Dato un capitale iniziale, un tasso di interesse annuo e il numero di anni in cui il capitale è stato depositato, calcolare il valore finale del capitale considerando l'interesse composto, cioè il capitale alla fine di ogni anno viene aumentato dell'interesse annuo e diviene il nuovo capitale per il calcolo dell'interesse l'anno successivo.
23. Dopo aver fatto inserire un numero intero positivo N all'utente, determinare qual è il massimo, con relativa molteplicità, tra N numeri dati in input uno alla volta. Se ad esempio N valesse 8 e la sequenza dei numeri fornita in input fosse 14, 7, 22, 19, 14, 8, 22, 11 la risposta sarebbe 22 con molteplicità 2.
24. Calcolare il valore del numero e di Eulero*, tramite la sommatoria che usa i reciproci del

* Il numero di Eulero è una costante matematica, come il più noto π, il cui valore approssimato alle prime 6 cifre è 2,71828.

fattoriale, come mostrato nella formula qua sotto, limitandosi al calcolo dei primi N termini, con N fornito dall'utente.

$$e := \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

25. Calcolare il valore del π utilizzando la formula di Leibniz come mostrata qua sotto, fermandosi al termine N-esimo, con N dato dall'utente.

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1} + \cdots = \frac{\pi}{4}$$

26. Dato in input un numero intero positivo n, verificare se è un quadrato perfetto, cioè se esiste un numero intero m tale per cui $m \cdot m = n$.
27. Progettare un algoritmo che scriva tutte le coppie di numeri che danno per prodotto n, con n fornito dall'utente, senza coppie ripetute.
28. Un numero intero positivo maggiore di 1 si dice *primo* se è divisibile solo per se stesso e per 1. Dato un numero n intero positivo ≥ 2 decidere se è primo o se non lo è, provando se ha altri divisori oltre all'1 e al numero stesso. Provare in un secondo momento a migliorare l'algoritmo per renderlo più efficiente, cioè fare in modo che esegua un numero minore di operazioni.
29. Dati in input il numeratore e il denominatore di una frazione, semplificarla ai minimi termini producendo in output i nuovi numeratore e denominatore.
30. Dato un intero n, trovare quanto vale la somma delle sue cifre. Se ad esempio il numero n fosse 123 il valore calcolato sarà 6.
31. Dati due numeri interi positivi n e m, calcolarne il prodotto utilizzando l'algoritmo del contadino russo. Una descrizione dell'algoritmo si può trovare all'indirizzo `https://www.wikihow.it/Svolgere-le-Moltiplicazioni-con-il-Metodo-del-Contadino-Russo`.
32. Dato un numero intero positivo n, stabilire se è divisibile per 11 utilizzando il criterio di divisibilità e non l'operatore modulo (%).
    Il criterio di divisibilità per 11 dice che un numero è divisibile per 11 se la differenza tra la somma delle cifre di posto pari e la somma di quelle di posto dispari è 0 o un multiplo di 11.

2. Hardware

Introduzione

Quanto detto nel capitolo precedente sull'informazione e gli algoritmi, trova il suo substrato fisico nei computer, che sono gli strumenti che permettono il trattamento automatico dell'informazione e l'effettiva *implementazione* di algoritmi astratti per la soluzione di un determinato problema.

Dal punto di vista di un programmatore la conoscenza dello schema fisico di un computer potrebbe eventualmente essere ignorata. Di fatto, però, avere almeno un'idea anche solo approssimativa di quali sono e che funzioni svolgono i componenti principali, può essere d'aiuto nella comprensione di alcuni aspetti della programmazione che altrimenti sfuggirebbero. In questo breve capitolo verrà quindi analizzato il computer nelle sue parti essenziali e di esse ne verrà data una descrizione non approfondita, ma che dovrebbe aiutare chi programma a farsi un'idea del funzionamento di un calcolatore[1].

1: A seconda del contesto, computer e calcolatore possono essere tranquillamente considerati sinonimi.

2.1. Architettura di un calcolatore

Il calcolatore deve la sua struttura concettuale alle idee di John Von Neumann, scienziato ungherese che negli anni Quaranta del Novecento propose uno schema[2], che da lui prende il nome e che viene indicato come *architettura di Von Neumann*, mostrato in Figura 2.1.

2: Essendo lo schema un concetto astratto piuttosto che un rigido diagramma tecnico, se ne possono trovare tante altre versioni, che comunque condividono le stesse idee fondamentali.

Essendo uno schema concettuale non rappresenta la struttura fisica di un computer quanto piuttosto le funzioni che si ritroveranno all'interno di un calcolatore indipendentemente dalla sua forma (PC, smartphone, calcolatrice programmabile, tablet, ecc.). Nei prossimi paragrafi si vedrà come ogni elemento di questo schema viene realizzato da un punto di vista fisico, senza entrare mai troppo nel dettaglio.

Il processore o CPU

Il *processore* (*CPU*, *Central Processing Unit* in inglese) è quello che spesso viene definito il *cervello* del computer: infatti è il dispositivo deputato al controllo della macchina e all'esecuzione delle

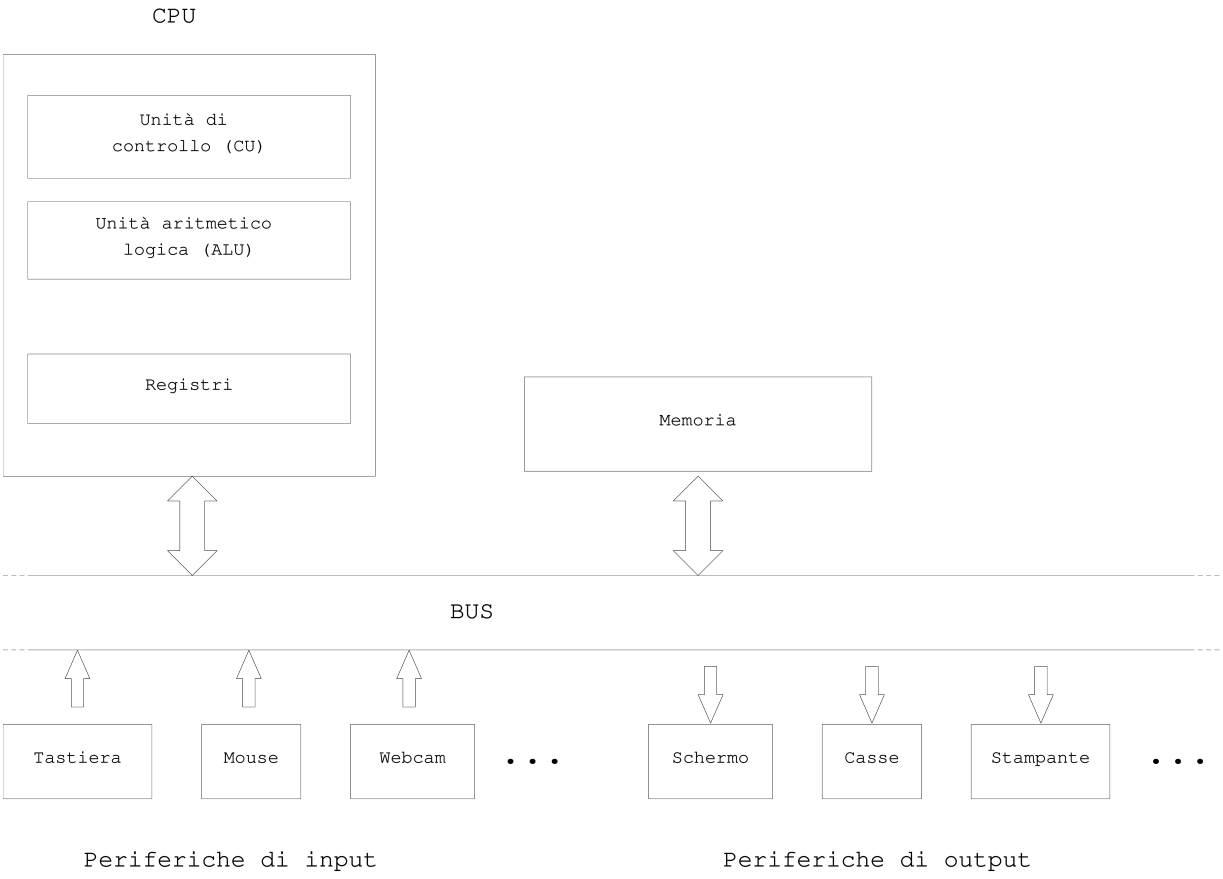


Figura 2.1.: Architettura di Von Neumann

istruzioni che gli vengono impartite dai programmi. Fisicamente si tratta di un circuito integrato, cioè un “pezzetto" di silicio, al cui interno sono stati fotoincisi dei componenti[3] e dei circuiti che nel complesso costituiscono la logica di funzionamento del dispositivo.

Per un programmatore il parametro più interessante che caratterizza un processore è la sua *frequenza*, misurata in Hz: la frequenza può essere pensata come il numero di operazioni che un processore riesce a eseguire in un secondo e quindi è direttamente proporzionale alla velocità del computer percepita dall’utente. Si può quindi dire che, a parità di tutti gli altri componenti, a una frequenza del processore più alta corrisponde un computer percepito come più veloce. I computer attuali hanno velocità dell’ordine dei Ghz[4] e quindi sono in grado di eseguire miliardi di operazioni in un secondo.

Questo fa si che i programmi che devono girare su un computer moderno non siano in generale affetti da problemi di performance e quindi il programmatore possa concentrarsi sul compito da risolvere e non sull’efficienza delle istruzioni che utilizzerà per risolverlo. Attenzione però che ci sono applicazioni (ad esempio i videogiochi) o contesti (macchine con processore dalla velocità

3: All’interno di un circuito ad altissima integrazione come un processore possono essere contenuti da qualche milione a qualche centinaia di milioni di transistor.

4: 1 Ghz (letto Giga Hertz) corrisponde a una frequenza di un miliardo di operazioni al secondo.

molto limitata, o macchine potenti ma che devono elaborare enormi quantità di dati) per le quali il problema delle prestazioni risulta cruciale, nel qual caso non è sufficiente risolvere il problema, ma è necessario che l'implementazione sia anche efficiente.

Nello schema di Figura 2.1 si può vedere che la CPU è composta dall'*unità di controllo* (*CU*, *Control Unit*) e dall'*unità aritmetico-logica* (*ALU*, *Arithmetic-Logic Unit*) e da dei *registri*. Senza entrare nel dettaglio si può dire che la CU è quella parte di circuiti che "coordina" le operazioni che si svolgono all'interno della CPU, la ALU si occupa di eseguire tutte le operazioni possibili sui dati e i registri sono dove vengono immagazzinati i dati su cui lavora la ALU e i risultati prodotti.

La memoria

Quando si parla di memoria in genere ci si riferisce alla memoria *RAM* (*Random Access Memory*, memoria ad accesso casuale), che è la parte di computer dove risiedono i programmi quando vengono eseguiti. La memoria serve appunto per registrare delle informazioni che rappresentano sia le istruzioni che dovrà eseguire la CPU sia i dati su cui esse lavoreranno.

Nel caso della memoria il parametro che più interessa un programmatore è la sua *dimensione*, che viene misurata in multipli del *byte*: il byte è formato da 8 *bit*, cioè contiene otto informazioni di tipo binario, che quindi possono rappresentare 256 stati diversi. Ogni byte, o meglio un insieme di byte, può contenere informazioni che possono rappresentare numeri, testi, immagini, ma anche le istruzioni che permettono di elaborare questi dati.

Le memorie odierne vanno dalle centinaia di MB (MegaByte, circa 10^6 byte) a qualche GB (GigaByte, circa 10^9 byte). Anche in questo caso, date le dimensioni abbondanti delle memorie odierne, il programmatore non avrà grossi problemi di ottimizzazione dello spazio nella realizzazione dei propri programmi, anche se vale lo stesso discorso fatto con la velocità del processore, che cioè ci sono contesti o applicazioni in cui è importante non sprecare spazio. In Figura 2.2 si può vedere una rappresentazione stilizzata di una memoria RAM.

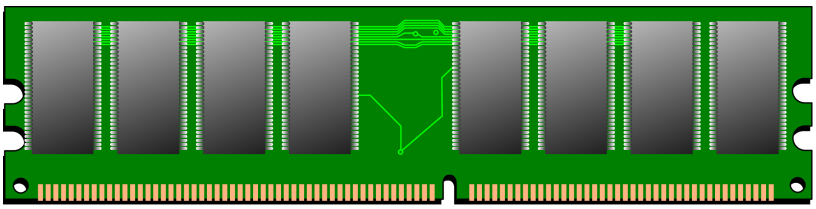


Figura 2.2.: Rappresentazione stilizzata di una memoria RAM

Un'altra caratteristica che definisce la memoria RAM è che può mantenere le informazioni solamente se è alimentata, in caso contrario vengono perse. Quando ad esempio il computer viene messo in *stand by*, vengono spenti una serie di componenti (ad esempio gli hard disk, la scheda grafica, ...), ma la memoria RAM viene mantenuta alimentata, in modo da poter ritornare velocemente alla situazione preesistente prima dello stand by, senza perdere nulla. Quando invece il computer viene messo in *ibernazione*, viene tolta l'alimentazione da tutti i componenti e il contenuto della RAM viene salvato sul disco fisso, che, come vedremo, è invece in grado di trattenere le informazioni anche in mancanza di alimentazione.

Infine dal punto di vista di un programmatore, soprattutto se usa linguaggi di programmazione che permettono l'accesso diretto alla memoria come il C++, la memoria può essere vista come una sequenza di "caselle", ognuna contenente un'unità di informazione, numerate partendo da 0 fino alla dimensione massima della memoria.

Memorie di massa

Con il termine memorie di massa vengono intesi tutti quei dispositivi che sono in grado di memorizzare l'informazione e di mantenerla per un tempo indefinito, anche se non vengono alimentati. Fra questi possiamo indicare i dischi fissi, sia magnetici (*HHD*, *Hard Disk Drive*) che a stato solido (*SSD*, *Solid State Drive*), le penne USB, le schede di memoria SD (*Secure Digital*), ma anche tecnologie adesso meno comuni perché più datate come i CD e i DVD, i dischi floppy, i nastri magnetici, alcune delle quali sono rappresentate in Figura 2.3

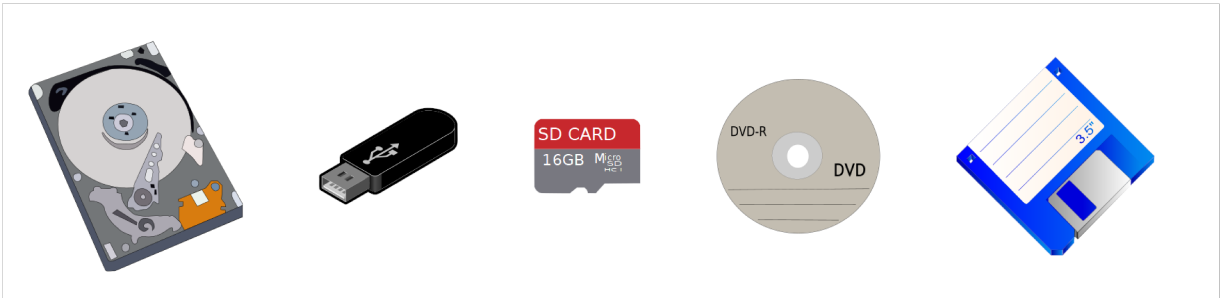


Figura 2.3.: Hard disk, penna USB, SD card, DVD, floppy disk

Anche in questo caso la caratteristica più significativa è la capacità di immagazzinare informazioni che, come per le memorie RAM, si misura in multipli del byte: i dischi fissi moderni hanno taglie dell'ordine delle centinaia di GB e in alcuni casi anche del TB (TeraByte).

I programmi sono in genere memorizzati nel disco fisso del computer, che può essere di tipo HHD o SSD, e quando vengono mandati in esecuzione vengono spostati dalla memoria di massa alla memoria RAM: si può dire che mentre i programmi su disco sono un'immagine *statica* contenente una serie di istruzioni, i programmi in esecuzione nella RAM sono un'immagine *dinamica*, che contiene i dati e il reale stato di esecuzione di un programma in quel particolare momento.

I dispositivi di input/output

I computer sono stati creati per essere utilizzati dagli umani e quindi hanno bisogno di un qualche meccanismo per comunicare e scambiare dati con il mondo esterno. I *dispositivi di input/output*[5] o, in forma abbreviata, di *I/O*, assolvono quindi questo scopo.

5: In italiano sono anche indicati come ingresso/uscita, ma la terminologia inglese è più utilizzata

Se nei primi anni di vita dei computer (intorno agli anni 40-50 del Novecento) i dispositivi di input/output erano piuttosto rozzi e di difficile utilizzo (ad esempio le schede perforate), adesso è possibile interagire con il computer utilizzando tastiere, visualizzando il risultato attraverso monitor a colori e nel futuro potrebbero comparire periferiche che renderanno ancora più semplice l'interazione con il computer.

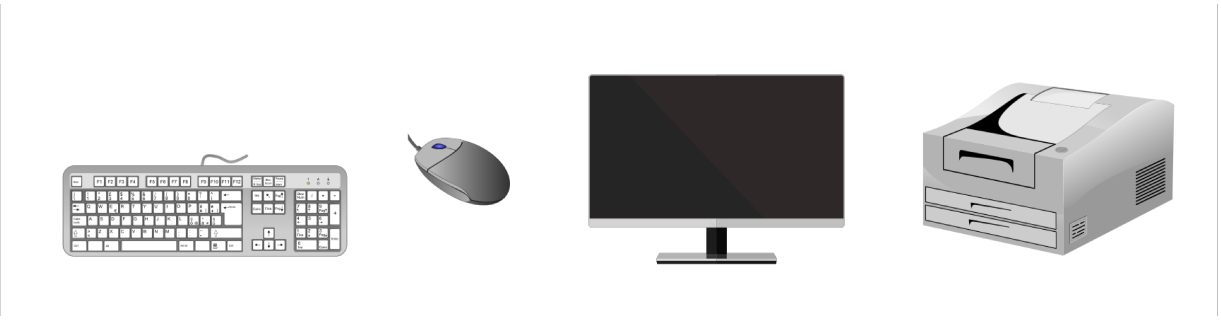


Figura 2.4.: Esempi di dispositivi di I/O: tastiera, mouse, monitor, stampante

Altri dispositivi

All'interno del computer, oltre ai componenti fondamentali appena descritti, esistono molte altre periferiche e dispositivi, alcuni necessari e altri non indispensabili, di cui si daranno brevi cenni, senza la pretesa di essere esaustivi:

- **scheda madre**: è una scheda elettronica, che solitamente occupa la "parete" di un computer, sulla quale sono alloggiati o comunicano tutti i componenti, alcuni inseriti all'interno di appositi slot (ad esempio il processore), altri collegati tramite cavi elettrici e altri ancora direttamente saldati sulla

sua superficie. Sono inoltre stampati sotto forma di “strade" in rame tutti i collegamenti elettrici che permettono di far comunicare tra di loro i vari componenti: riprendendo l'analogia con il corpo umano, si potrebbe pensare alla scheda madre come una specie di sistema nervoso del computer

- **scheda grafica**: è una scheda che monta un processore specializzato nelle operazioni grafiche (*GPU*, *Graphics Processing Unit*), spesso con una memoria separata dalla RAM del computer: il suo scopo è quello di gestire la parte di disegno dei programmi, in particolare quelli con marcate esigenze prestazionali, come ad esempio i videogiochi. In alcuni computer, in genere i portatili, la scheda grafica non è separata dalla scheda madre, ma è un chip grafico saldato direttamente su di essa oppure integrato direttamente nella CPU.
- **memoria ROM** (Read Only Memory): è la memoria dove viene memorizzato il *BIOS* (*Basic Input Output System*) del computer, cioè quel programma di basso livello che serve a inizializzare correttamente il computer e che contiene istruzioni che vengono eseguite prima dell'avvio del sistema operativo. È una memoria che conserva le informazioni anche se non è alimentata e da un po' di anni è possibile aggiornarne il contenuto[6] per installare versioni più recenti del BIOS.
- **scheda di rete**: anche in questo caso, come per la scheda grafica, può essere una scheda separata oppure essere integrata su scheda madre (nei computer recenti è questa la soluzione più diffusa). Ha lo scopo di permettere al computer di collegarsi a una rete attraverso un cavo apposito. Ormai quasi tutti i computer hanno integrato al loro interno anche un chip per connettersi alle reti wireless.
- **scheda sonora**: spesso il chip per la gestione dell'audio è integrato su scheda madre, tranne dei casi dove l'elaborazione sonora ricopra una certa importanza, nel qual caso la scheda sonora è separata e permette maggiore qualità e funzionalità

6: In questo caso il termine più corretto sarebbe *EEPROM* (*Electrically Erasable Programmable Read-Only Memory*), in quanto la modifica avviene direttamente all'interno del computer tramite circuiti che ne modificano le informazioni contenute per aggiornarle con versioni più recenti.

2.2. Esercizi

1. Elencare le caratteristiche dei principali dispositivi (CPU, RAM, disco fisso, . . .) che compongono il proprio computer. La maggior parte delle informazioni possono essere ottenute direttamente dal sistema operativo, oppure esistono programmi appositi, come HWiNFO o CPU-Z, che forniscono informazioni approfondite su vari componenti del computer.
2. Indicare altre periferiche di IO oltre a quelle indicate nel testo.
3. Creare un documento digitale contenente delle immagini, fatte con il proprio smartphone o reperite su Internet, dei componenti che sono stati descritti in questo capitolo, in modo da aver presente come si presentano realmente. Non è importante raccogliere immagini di ogni componente possibile, limitarsi pure a quelli fondamentali, possibilmente quelli presenti nel proprio computer.
4. Fare una sintesi scritta delle principali informazioni contenute in questo capitolo, studiando prima con attenzione ogni paragrafo in modo da poter scrivere senza aver necessità di consultare il libro.

Il linguaggio C++ 3.

Introduzione

Dopo aver visto nel capitolo precedente cosa significa *algoritmo* e aver utilizzato un formalismo grafico, i diagrammi di flusso, per trasformare delle idee risolutive in programmi che accettano input e forniscono delle risposte, adesso si vedrà come gli stessi concetti possano essere comunicati a un computer utilizzando un linguaggio apposito, in questo caso il C++.

L'esigenza di utilizzare un linguaggio di programmazione nasce dal fatto che i computer sono macchine in grado di essere istruzioni in modo preciso, rapidissimo, senza stancarsi mai o fare errori, ma queste istruzioni sono poche e definite dall'hardware del computer, quindi è necessario tradurre qualsiasi cosa si voglia far eseguire a un computer in una sequenza di operazioni che sia in grado di eseguire.

Se inizialmente queste istruzioni venivano fornite al computer direttamente in binario, in una forma direttamente eseguibile, ben presto ci si è resi conto che lavorare in binario era un compito molto faticoso perchè troppo distante dal modo di pensare delle persone, molto più astratto e di alto livello. Questo ha fatto sì quasi da subito si sia cercato di evitare di scrivere in questo modo, ma piuttosto di creare dei linguaggi più semplici da gestire, il cui contenuto fosse successivamente tradotto in codice binario: è così iniziata la storia dei linguaggi di programmazione, che continua ancora oggi, con la continua *invenzione* di nuovi linguaggi, adatti ai nuovi contesti che nascono con l'avanzare della tecnologia.

3.1. Il linguaggio C++

Il linguaggio C++ viene creato all'inizio degli anni '80 da Bjarne Stroustrup negli AT&T Bell Labs, cercando di fondere le caratteristiche del C, al tempo linguaggio con ampia diffusione, ottima velocità e portabilità, con i nuovi concetti della *programmazione orientata agli oggetti*, prendendo come modello il linguaggio Simula, nato per essere *object-oriented*, ma troppo lento per un utilizzo pratico. Da allora si sono succedute una serie di versioni, standardizzate dall'ISO (International Standard Organization), partendo dalla prima versione del 1998, fino ad arrivare alla versione C++20, l'ultima definita al momento della stesura di questi appunti.

Definizione di linguaggio di programmazione

I linguaggi di programmazione altro non sono che un insieme di regole, generalmente raccolte in grossi documenti, che definiscono senza ambiguità il significato di ogni parola del linguaggio (lessico), i modi con cui si possono comporre delle «frasi» con quelle parole (sintassi) e il significato che hanno per un computer quelle frasi (semantica).

Il processo di traduzione che consente di passare dal linguaggio di programmazione al codice binario (o *codice macchina*) può essere effettuato utilizzando due metodologie distinte:

- usando un *interprete*, cioè un programma che prende in input il *codice sorgente*[1] e lo esegue linea per linea. Tra le conseguenze più evidenti di questa metodologia si possono elencare:
  - è possibile testare immediatamente ogni modifica fatta al codice, in quanto basta darlo *in pasto* all'interprete
  - è necessario avere un interprete sulla macchina sulla quale si vuole far girare il programma
  - il programma può girare su qualsiasi architettura[2] di cui si disponga di un interprete per quel linguaggio
  - generalmente, una certa lentezza di esecuzione, causata appunto dal fatto di dover tutte le volte reinterpretare il programma[3].

  Esempi di linguaggi interpretati sono Python, PHP, Javascript.
- usando un *compilatore*, cioè un programma che prende in input il *codice sorgente* e lo trasforma in codice macchina direttamente eseguibile dal computer. Anche in questo caso il processo di compilazione ha le proprie caratteristiche:
  - ogni modifica al codice richiede la *ricompilazione*, quindi, a seconda delle dimensioni del codice che è necessario ricompilare[4], possono esserci dei tempi d'attesa anche lunghi
  - non è necessario avere un compilatore sulla macchina sulla quale si vuole far girare il programma, in quanto il programma compilato è già in codice macchina e può girare in modalità *stand-alone* senza dipendenze da altri software
  - il programma può girare su qualsiasi architettura cui si disponga di un compilatore per quel linguaggio, a patto di ricompilare il sorgente
  - generalmente, massima efficienza di esecuzione, poichè il programma è in codice macchina e viene eseguito

1: Con codice sorgente, o *source code*, si intende il testo scritto da un programmatore in un certo linguaggio di programmazione.

2: Per architettura in questo contesto si intende l'insieme macchina fisica - sistema operativo.

3: Nella realtà esistono tecniche avanzate per migliorare la velocità di esecuzione, ma non verranno qui prese in considerazione.

4: Anche in questo caso esistono tecniche che permettono di ridurre il tempo d'attesa.

direttamente sulla macchina fisica, senza necessità di strati di interpretazione intermedi

Esempi di linguaggi compilati sono C/C++, Rust, Go.

In Figura 3.1 si può vedere tutto il processo di compilazione di un programma scritto in C++:

1. tramite un editor oppure un IDE (Integrated Development Environment, strumento di cui si parlerà a breve), lo sviluppatore scrive il programma seguendo le regole del C++
2. quanto scritto, più eventuali *header* di libreria[5], vengono passati al *preprocessore*, che effettua delle semplici operazioni per creare un unico file che verrà poi preso in ingresso dal compilatore
3. il compilatore applica le regole del linguaggio C++ a quanto scritto e lo trasforma in una serie di codici binari in modo che possano essere eseguiti dal computer
4. il *linker* effettua l'ultima fase, mettendo insieme il codice binario prodotto al passo precedente con eventuali altri codici binari, dove sono racchiuse funzionalità necessarie al programma. In uscita da questa fase si ha il programma eseguibile, cioè quel file binario che potrà essere caricato nella memoria del computer ed essere mandato in esecuzione.

5: Un header è un file, sempre scritto in C++, che serve al compilatore per verificare la correttezza di funzioni di libreria eventualmente usate nel codice sorgente. Questo discorso verrà approfondito nel capitolo che riguarda le funzioni.

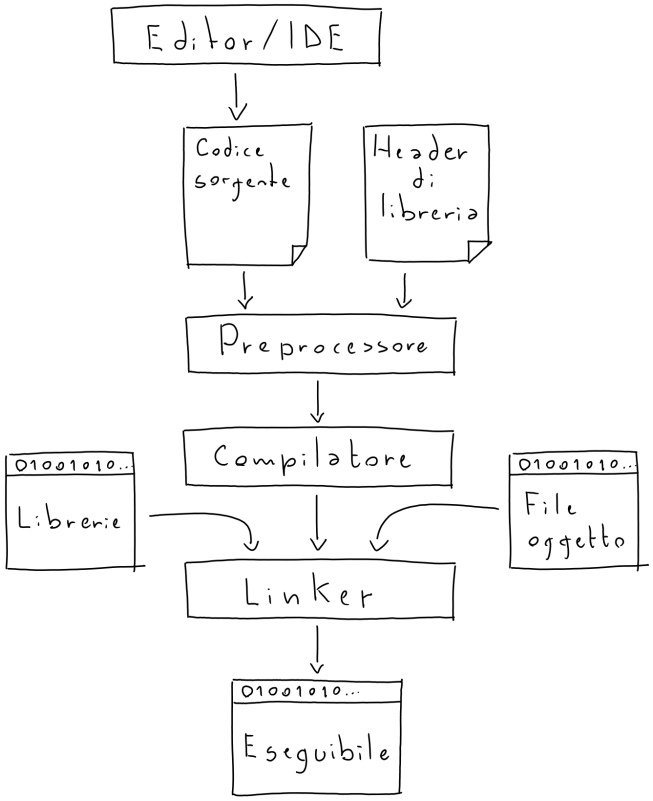


Figura 3.1.: Catena di sviluppo di un programma in C++.

A questo punto dovrebbe essere chiaro che l'attività di sviluppo software sia prevalentemente un'attività che coinvolge il pensiero e che si traduce nella scrittura di codici in un particolare linguaggio,

simile a quanto fanno gli scrittori, con la rilevante differenza che, mentre quanto scritto da uno scrittore produce una serie di effetti nella mente di chi legge, quanto scritto da un programmatore produce effetti all'interno di un computer.

Come gli scrittori usano carta e penna o, più recentemente, computer o tablet, anche il programmatore ha i propri strumenti, in particolare il già citato IDE, Integrated Development Environment, o *ambiente di sviluppo integrato*. Questo software specifico aiuta nella scrittura dei programmi, in particolare:

- colora il codice (*syntax-hilighting*) per renderlo più intelleggibile e in modo che sia più evidente l'eventuale presenza di errori
- semplifica il processo di compilazione, composto da molti passaggi come visto in Figura 3.1, solitamente riducendolo alla pressione di un tasto
- gestisce progetti complessi composti da molti file
- suggerisce come completare parti di codice e velocizza la scrittura permettendo di svolgere operazioni complesse attraverso delle scorciatoie da tastiera

Siccome il codice sorgente è solo testo, basterebbe anche il semplice Blocco Note per poter scrivere un programma, però non conterrebbe nessuna delle funzionalità citate, rendendo l'attività di programmazione molto più faticosa.

3.2. Istruzioni elementari in C++

Nei prossimi paragrafi verranno mostrate le istruzioni di base del linguaggio C++, necessarie per poter scrivere un programma funzionante e che hanno quasi sempre dei diretti paralleli con quanto visto in precedenza nei diagrammi di flusso e, dove possibile, verrà mostrato questo parallelismo.

Un primo programma

Dopo aver spiegato come funziona un linguaggio di programmazione, è il momento di mostrare un primo, semplicissimo programma.

Listing 3.1: Primo programma in C++.

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello world!" << std::endl;
5      return 0;
6  }
```

Anche in un programma così elementare, che si limita a stampare a video la stringa «Hello world!», ci sono una serie di istruzioni che sono necessarie per permettere al compilatore di produrre un codice eseguibile, alcune delle quali verranno spiegate in seguito, ma che non possono essere omesse se si vuole ottenere un sorgente valido.

In generale non sempre sarà possibile spiegare il significato esatto di ogni istruzione, a volte si preferirà rimandare a un secondo momento alcuni concetti che altrimenti risulterebbero confusi o troppo astratti.

Si noterà che le righe 1, 3, 5 e 6 saranno sempre presenti in ogni programma funzionante e la parte scritta dal programmatore, lunga da poche righe a diverse decine o centinaia, verrà inserita dove in questo esempio si trova l'istruzione `std::cout << "Hello world!" << std::endl;`.

Un'altra cosa che è bene evidenziare fin da subito e che risulterà ancora più chiara nel seguito del testo, è che la scrittura di un programma prevede che il codice non sia tutto allineato a sinistra, come succede in un testo normale, ma segua dei rientri, chiamati *indentazioni*, con lo scopo di rendere più semplice da comprendere la struttura logica del programma.

La regola generale è quella di allineare tutte le istruzioni appartenenti allo stesso blocco di codice e di rientrare di una certa quantità di spazi, solitamente 4, blocchi di codice se sono contenuti all'interno di altri. L'IDE aiuta nell'ottenere questo risultato, posizionando il cursore nella posizione giusta ogni volta che si va a capo, deve comunque essere cura del programmatore sviluppare uno stile di scrittura ordinato per ottenere una buona indentazione generale.

Il compilatore C++ ignora l'indentazione, che quindi è pensata come un supporto alla leggibilità per i programmatori. Per evidenziare questo fatto si può vedere Figura 3.2 che è un esempio estremo di codice C++ corretto e funzionante, con però un'indentazione pessima (seppure decisamente artistica), che il compilatore non avrebbe problemi a digerire, e che però metterrebbe in difficoltà un programmatore che provasse a capire cosa fa quel programma[6].

6: L'esempio è tratto da un competizione per scrivere codice funzionante il più difficile da capire possibile e in questo caso prende in input un file immagine, in tre diversi formati, e ne fa il downsampling

Tipi e variabili

Come già visto con Flowgorithm, prima di poter scrivere un algoritmo che utilizzi delle variabili per i suoi scopi (input, elaborazione o output), è necessario dichiararle. Per la dichiarazione bisogna indicare il *tipo* e il *nome*. Per quanto riguarda il tipo ci si limiterà soltanto a quelli fondamentali, elencati nella Tabella 3.1.

Figura 3.2.: Esempio di indentazione *particolare* (IOCCC 2011).

Tabella 3.1.: Tipi di dati fondamentali in C++

Tipo	Significato
`bool`	Variabile booleana, può assumere i valori `true` o `false`
`char`	Carattere a 8 bit, per memorizzare caratteri ASCII, internamente un valore intero compreso tra 0 e 255
`int`	Valore intero con segno, la lunghezza dipende dall'ambiente, tipicamente a 32 bit, quindi valori compresi tra circa -2 miliardi e +2 miliardi
`float`	Valore in virgola mobile a 32 bit secondo lo standard IEEE 754
`double`	Valore in virgola mobile a 64 bit secondo lo standard IEEE 754

Per quanto riguarda i nomi esistono delle regole e delle convenzioni. Per quanto riguarda le regole, basterà seguire queste:

- sono formati[7] dai caratteri della lingua inglese, maiuscoli o minuscoli, da cifre, che però non possono apparire alla prima posizione e dal carattere *underscore*, cioè il simbolo _
- i nomi sono *case-sensitive*, quindi `altezza` e `Altezza` sono due nomi differenti
- non possono esserci due variabili nello stesso *scope*[8] con lo stesso nome.

Le convenzioni sono invece delle scelte fatte dai programmatori di un linguaggio, che si sono sedimentate nel tempo poiché ritenute sensate e che sono perlopiù condivise da tutti i programmatori di quel linguaggio:

- i nomi di variabili usano lettere minuscole, a meno che non rappresentino nomi che nella realtà sono sempre scritti in

7: Esistono altre regole per i caratteri validi, ma questo sottoinsieme è sufficiente e evita confusione.

8: Nel capitolo delle funzioni verrà chiarito cosa si intende con *scope* (contesto di visibilità).

maiuscolo, ad esempio `MCD` per il Massim Comun Divisore o `CAP` per il codice di avviamento postale
- quando una variabile, per chiarezza, si vuole che sia composta da due o più parole, ognuna di queste è separata dalla precedente con un *underscore*, come ad esempio `costo_totale` o `biglietti_venduti`

Si ricorda che il nome deve aiutare il programmatore a ricordare cosa rappresenta la variabile nel contesto del problema che si vuole risolvere e a rendere il codice più facile da leggere e interpretare, anche a distanza di tempo dalla prima scrittura.

Per dichiarare una variabile la sintassi si può vedere nel listato 3.2:

Listing 3.2: Dichiarazione di variabili.

```
1  int main() {
2      int altezza;
3      int base, contatore = 0;
4      float perimetro, area;
5      return 0;
6  }
```

In generale bisogna prima indicare il *tipo* della variabile e successivamente il nome, il tutto terminato dal punto e virgola, che è il terminatore delle istruzioni.

Sono consentite alcune varianti, presenti nel codice 3.2, che sono:

- è possibile dichiarare più nomi di variabili sulla stessa riga, separandole con la virgola, come si può vedere alla riga 3 e 4, a patto che siano dello stesso tipo
- ad una variabile può essere assegnato un valore al momento della dichiarazione, come si vede per la variabile `contatore` alla riga 3. In questo caso l'assegnamento prende anche il nome di *inizializzazione*, poiché stabilisce il valore iniziale della variabile[9].

9: In C++, una variabile non inizializzata ha un valore indefinito.

Conversione di tipo e casting

I tipi numerici, come già visto, possono essere divisi in due famiglie: gli *interi* e i *numeri con la virgola* (*floating-point*). La differenza tra queste due famiglie è sostanziale e deriva dal fatto che in hardware ci sono due sezioni distinte, la ALU e la FPU[10], che si occupano rispettivamente dei calcoli tra interi e dei calcoli tra numeri in virgola mobile. Senza entrare nei dettagli, questa differenza comporta delle conseguenze nel momento in cui si provano a combinare in un'operazione due variabili di tipo diverso. La regola generale prevede la promozione dei tipi:

10: **A**rithmetic **L**ogic **U**nit e **F**loating **P**oint **U**nit.

Promozione dei tipi di variabili

Si ha promozione di tipo quando vengono assegnate o combinate, tramite un operatore matematico, due variabili numeriche di tipo diverso: in questo caso la variabile di tipo «meno preciso» viene promossa a quella di tipo «più preciso»[11].

11: I termini più preciso e meno preciso in questo contesto rischiano di essere male interpretati: senza approfondire troppo, sia gli interi che i numeri `float` hanno 32 bit e quindi la stessa *potenza rappresentativa*, la differenza risiede in cosa possono rappresentare e in come lo rappresentano.

Limitandosi ai tre tipi numerici principali, gli `int`, i `float` e i `double`, le regole sono le seguenti:

Operazione	Promozione
`int` op `float`	`int` ⟶ `float`
`int` op `double`	`int` ⟶ `double`
`float` op `double`	`float` ⟶ `double`

Alcuni esempi possono essere visti nel codice 3.3:

Listing 3.3: Promozione di tipo.

```
1  int main() {
2      int a = 7;
3      float b = 7.5;
4      double c = 7.5;
5      float r = a + b; //14.5
6      double s = a + c; //14.5, ma double
7      double t = b + c; //15.0, sempre double
8      return 0;
9  }
```

L'effetto di queste promozioni è quello di utilizzare la FPU piuttosto che la ALU per fare i calcoli che coinvolgono le variabili oggetto dell'operazione. Bisogna però porre attenzione al fatto che le operazioni sono **sempre** svolte tra due operandi, una di seguito all'altra e questo potrebbe portare a situazioni inaspettate, come quelle illustrate nel codice 3.4

Listing 3.4: Casting.

```
1  int main() {
2      int a = 7;
3      float b = 7.5;
4      float r = b + a / 2; // 10.5 anzichè 11
5      std::cout << "r = " << r << std::endl;
6      r = b + (float)a / 2; // 11, con casting esplicito
7      std::cout << "r = " << r << std::endl;
8      r = b + a / 2.0; // 11, con promozione
9      std::cout << "r = " << r << std::endl;
10     return 0;
11 }
```

Nella riga 4, in effetti, il risultato atteso sarebbe di trovare il valore 11 nella variabile r, cosa che invece non avviene, in quanto il valore memorizzato risulta 10.5 e non 11. Per comprendere perché succede, è necessario analizzare la sequenza di operazioni che vengono eseguite alla riga 4 e che sono un assegnamento, una somma e una divisione. L'ordine con cui sono eseguite è il seguente:

1. la divisione è la prima operazione ad essere eseguita, in quanto quella con la priorità più alta. Entrambi gli operandi, la variabile `a` e il numero 2, sono interi e quindi l'operazione viene svolta dalla ALU e il risultato, conseguentemente, è intero. Siccome il valore in `a` è 7, il risultato della divisione è 3 e la parte dopo la virgola viene «persa».
2. successivamente viene eseguita la somma, che avendo come operando una variabile **`float`** e un intero, il risultato della precedente divisione, effettua una promozione e svolge l'operazione nella FPU, ma ormai la parte decimale della divisione è stata persa
3. infine avviene l'assegnamento, ma ormai il risultato è 10.5 e così viene memorizzato nella variabile `r`.

Per risolvere questo problema si possono seguire due strade, una più generale e una specifica:

1. come si può vedere alla riga 6, viene applicato l'operatore di *casting*, che consiste nell'indicare prima di una variabile il tipo a cui la si vuole convertire, in questo caso **`float`**, racchiudendolo tra parentesi tonde. In questo esempio la variabile `a`, da intera viene convertita in **`float`** e, quindi, la divisione sarà tra un **`float`** e un **`int`**, producendo come risultato 3.5
2. l'altra strada, che si può vedere alla riga 8, è quella di rappresentare il numero 2 come numero con la virgola, quindi 2.0, ottenendo lo stesso risultato, essendo ancora una divisione tra un intero e un **`float`**. Chiaramente un approccio del genere può essere fatto solo dove l'operazione comprenda un operando in forma di numero, mentre il primo approccio può essere applicato in qualsiasi situazione.

Le variabili costanti

Sebbene il titolo di questo paragrafo possa sembrare contraddittorio (se qualcosa è *variabile* come è possibile che sia anche *costante*?), lo scopo di questo meccanismo del linguaggio è di poter «dare un nome» a un valore, in modo da rendere il codice più leggibile.

Nel banale esempio riportato nel listato 3.5, si può vedere che viene utilizzata la parola chiave **`const`** per indicare che la variabile `TASSA_SUCCESSIONE`, di tipo **`float`**, è una variabile costante. Le due conseguenze principali di questa dichiarazione sono:

- il codice diventa più leggibile in quanto, al posto del numero 0.06 si utilizza TASSA_SUCCESSIONE, che chiarisce il significato del numero

- se per errore si cercasse di assegnare un valore a una variabile costante, il compilatore lo segnalerebbe e si rifiuterebbe di compilare il programma

Si può inoltre notare che il nome sia composto solo da maiuscole, più eventualmente un *underscore* per separare più parole: questa è una convenzione molto diffusa, che permette a un programmatore che legga codice non scritto da lui, di capire subito che quella variabile è in realtà costante. Infine la posizione dove viene fatta la dichiarazione è insolita rispetto a quanto visto finora: il motivo è che una dichiarazione fatta al di fuori del `main` ha visibilità *globale*, cioè verrà vista in ogni parte del programma, caratteristica utile per una variabile che rappresenta un valore che può trovare utilizzo in svariate parti del codice[12].

12: Nel prossimo capitolo sulle funzioni risulterà chiaro come mai un programma possa essere composto da più parti.

Listing 3.5: Dichiarazione di variabile costante.

```
1  #include <iostream>
2
3  const float TASSA_SUCCESSIONE = 0.06;
4
5  int main() {
6      float capitale = 100000;
7      std::cout << "La tassa da pagare sará " << capitale
       * TASSA_SUCCESSIONE << " euro";
8      TASSA_SUCCESSIONE = 0.08; //istruzione non valida,
       errore in compilazione
9      return 0;
10 }
```

Operatore aritmetici, relazionali e logici

Per gli operatori aritmetici, relazionali e logici valgono le regole già indicate nell'Appendice A. In C++ esistono però alcune altre notazioni che sono di uso molto comune e che è utile conoscere, sebbene non siano strettamente necessarie. Il listato 3.6 ne mostra alcuni esempi.

Listing 3.6: Forme idiomatiche degli operatori.

```
1  #include <iostream>
2
3  int main() {
4      int a = 3, b = 7, c;
5      a++; //equivalente a a = a + 1;
6      c = a + b;
7      c += 5; // equivalente a c = c + 5;
8      c /= 4; // equivalente a c = c / 4;
9      return 0;
10 }
```

La prima è l'operatore di incremento che si vede alla riga 5 indicato con il simbolo ++: questa notazione indica che la variabile verrà incrementata di un'unità[13] ed è quindi equivalente a `a = a + 1`.

13: Il nome C++ deriva proprio dal fatto che inizialmente è stato pensato come una versione del C incrementata.

Esiste anche le notazione `--` che ha l'ovvio significato di decrementare di un'unità. Di entrambe esiste anche la versione con *preincremento* (o *predecremento*), in cui gli operatori vengono messi prima del nome di variabile: sebbene l'effetto non sia esattamente lo stesso, nel caso di istruzioni isolate come quella di riga 5, non ci sono differenze tra i due modi.

La seconda istruzione idiomatica è la forma contratta degli operatori matematici, che si può trovare sia alla riga 7 che alla riga 8: in questo caso la notazione indica che la variabile sulla sinistra verrà aumentata del valore indicato alla destra, nel caso della riga 7. Quindi la scrittura è equivalente a `c = c + 5` e al posto dell'addizione si può utilizzare qualsiasi altro simbolo aritmetico, con lo stesso significato, come si può vedere alla riga 8 con la divisione.

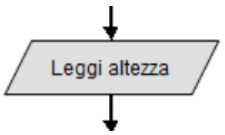
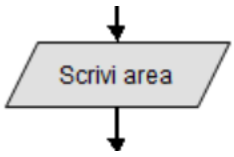
Istruzioni di input/output

Come visto nel capitolo sugli algoritmi, ogni programma può essere visto come composto da input, elaborazione e output. Per i programmi che verranno sviluppati nel resto del testo, che saranno di tipo *testuale* o a *console*, l'input sarà costituito dai caratteri inseriti da tastiera dall'utente, mentre l'output sarà un insieme di caratteri che verranno visualizzati nella console testuale.

In C++ sia l'input che l'output non sono istruzioni di base del linguaggio, ma funzionalità implementate attraverso una libreria standard, che in questo caso è la `iostream` e che viene inclusa nel programma attraverso la prima riga che compariva[14] anche negli esempi precedenti, tramite una direttiva di inclusione, `#include <iostream>`.

14: A volte nei listati viene omessa per brevità di scrittura, ma in codici reali che facciano uso di input/output deve essere presente.

Tabella 3.2.: Equivalenza input/output nei diagrammi di flusso

Istruzioni C++	Diagrammi di flusso
`std::cin >> altezza;`	
`std::cout << area << std::endl;`	

Come si può vedere nella Tabella 3.2 ci sono due istruzioni distinte, che possono essere interpretate nel seguente modo:

- per l'input esiste un oggetto particolare, `std::cin`, che può essere pensato come la tastiera, da cui arriverà l'input. Se

pensato in questo modo i due simboli consecutivi di maggiore, `>>`, indicano la direzione dalla tastiera verso la variabile, in questo esempio `altezza`, dove verrà memorizzato il valore inserito dall'utente

- per l'output esiste un oggetto analogo, `std:cout`, che può essere pensato come la tastiera, a cui andrà l'output. Anche in questo caso i due simboli consecutivi di minore, `<<`, indicano la direzione che dalla variabile va verso lo schermo, per indicare che il contenuto della variabile verrà mostrato a video. In questo esempio si può inoltre notare che possono essere utilizzate più concatenazioni di espressioni da mostrare a video ed è presente un oggetto speciale, `std::endl`, che rappresenta il *ritorno a capo* e che quindi sposterà il cursore di stampa nella riga sottostante a quella corrente.

Commenti

Quando si scrive un programma può sorgere la necessità di inserire delle *annotazioni*, con lo scopo di rendere più chiaro quanto scritto o documentare scelte che altrimenti non sarebbero facilmente comprensibili. Per questo i linguaggi di programmazione permettono di inserire dei *commenti*, cioè del testo, generalmente in linguaggio naturale, utilizzando delle notazioni opportune. Come già visto nel listato 3.6, si possono inserire dei commenti all'interno del codice sorgente, usando una coppia di *slash* (`//`): tutti i caratteri alla loro destra fino alla fine linea saranno ignorati dal compilatore e quindi potranno essere quello che si vuole.

Esiste inoltre un altro tipo di commento, generalmente chiamato commento *multilinea*, che solitamente viene usato quando le informazioni da inserire sono numerose e si espandono su più linee. In questo caso i caratteri di inizio commento saranno `/*` e tutti i caratteri inseriti fino alla coppia `*/`, saranno ignorati dal compilatore. Un esempio di entrambi i tipi di commenti possono essere visti nel listato 3.7

Listing 3.7: Commenti a linea singola e multilinea.

```
1  //Questo è un commento su linea singola
2
3  /*
4      Questo invece è un commento
5      multilinea
6  */
```

3.3. Le strutture di controllo

Qualsiasi linguaggio di programmazione contiene al suo interno delle parole chiave che permettono di implementare le struttu-

re di *sequenza*, *selezione* e *iterazione*. Mentre la sequenza deriva semplicemente dalla scrittura su righe consecutive di istruzioni, la selezione e l'iterazione permettono di modificare il normale flusso del programma, consentendo al programmatore di scrivere qualsiasi algoritmo. Nei due prossimi paragrafi si metterà in evidenza soltanto la corrispondenza con le equivalenti strutture nei diagrammi di flusso.

La selezione

La selezione permette di modificare il flusso di controllo, eseguendo o meno alcune istruzioni in base alla verità o falsità di una certa condizione. Negli esempi in Tabella 3.3 si possono trovare le due differenti forme base della selezione:

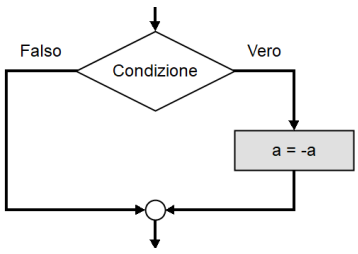
- l'`if` semplice, in cui un'istruzione viene eseguita o meno in base alla condizione
- l'`if-else`, dove esistono due rami di esecuzione alternativi, dei quali solo uno verrà eseguito: viene così garantito che sicuramente uno dei due rami verrà eseguito e, allo stesso tempo, che non sarà possibile vengano eseguiti entrambi.

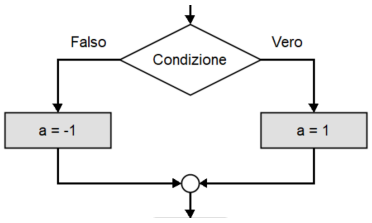
Una caratteristica comune sia alla selezione che all'iterazione, come si vedrà in seguito, è che le istruzioni che vengono condizionate fanno parte di un *blocco* delimitato dalle parentesi graffe. In questi esempi ogni blocco contiene una sola istruzione, ma il numero di istruzioni contenute può essere del tutto arbitrario, l'importante è che ad ogni parentesi graffa aperta ne corrisponda una chiusa[15].

15: A questo punto si potrebbe notare che anche dopo l'istruzione `main` del primo programma d'esempio comparivano delle parentesi graffe, e in effetti delimitano il blocco di codice che è il programma che verrà eseguito.

Tabella 3.3.: Equivalenza della selezione nei diagrammi di flusso.

Istruzioni C++	Diagrammi di flusso

`if (Condizione){`
`  a = -a;`
`}`
	

`if (Condizione){`
`  a = 1;`
`} else {`
`  a = -1;`
`}`
	

Combinando opportunamente questo semplice costrutto si possono realizzare strutture per gestire casi più complessi, in cui non la

scelta non si limita solamente alla verifica di una condizione, ma risulta più articolata. Solo a titolo di esempio:

- una variabile che può assumere un insieme limitato di valori interi, ad ognuno dei quali corrisponde l'esecuzione di una certa azione
- l'appartenenza o meno di un certo valore a degli intervalli specifici
- un insieme di variabili i cui valori hanno delle relazioni e in base a queste relazioni vengono eseguite alcune azioni piuttosto che altre.

A seconda di come vengono composte le condizioni si possono creare delle selezioni *annidate* oppure in *cascata*, come si può vedere nella Tabella 3.4. I due esempi potrebbero sembrare del tutto simili, se non uguali, almeno come effetto: leggendo il codice o guardando i diagrammi di flusso si dovrebbe intuire che lo scopo di questo frammento di codice è quello di trovare il minore tra tre numeri e stamparlo a video. Il diagramma di flusso però evidenzia bene che nel primo caso verrà stampato un solo numero qualsiasi siano i valori di `a`, `b` e `c`, mentre con le selezioni in cascata si potrebbe arrivare al caso limite in cui verrebbe stampato lo stesso numero per tre volte, quando i valori di `a`, `b` e `c` coincidono. In questo specifico esempio quindi una selezione annidata risulta migliore di una selezione in cascata, ma in generale la scelta tra una tipologia e l'altra dipende da cosa si vuole ottenere: quindi non sono da scegliere o scartare a priori, ma a seguito di un'analisi del problema che si intende risolvere.

L'iterazione

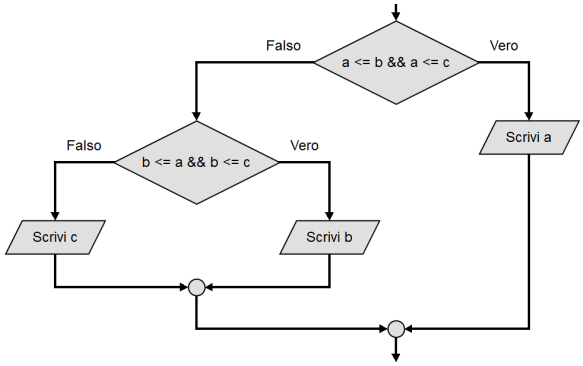
Come già discusso nel capitolo precedente, nei diagrammi di flusso esistono tre differenti modi di implementare il costrutto di iterazione e anche nel linguaggio C++ si ritrovano queste tre tipologie[16], indicate con le parole chiave **`while`**, **`for`** e **`do-while`**. Nella Tabella 3.5 ne viene mostrata la sintassi: per quanto riguarda l'utilizzo valgono le considerazioni già fatte nel capitolo precedente che qui verranno brevemente riassunte:

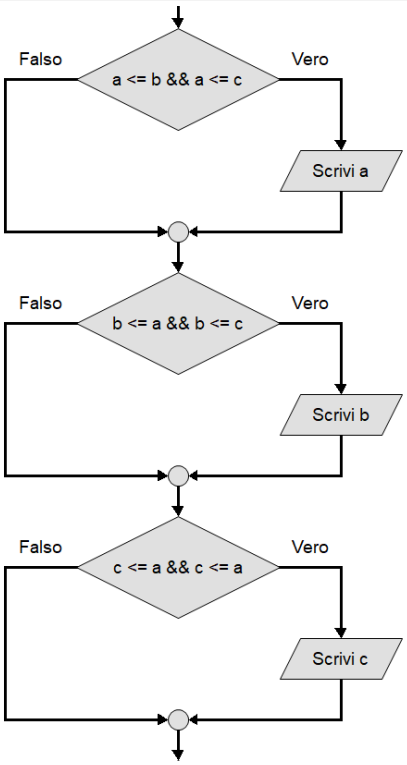
- il **`while`** implementa un ciclo **indeterminato**, va quindi usato quando non è noto il numero di ripetizioni che verranno fatte, ma si è a conoscenza della condizione che lo porterà a terminazione
- il **`for`** implementa invece un ciclo **determinato**, dove viceversa è noto a priori il numero di ripetizioni che dovranno essere eseguite per arrivare alla soluzione del problema
- il **`do-while`** è simile al **`while`**, con la differenza che il corpo del ciclo viene eseguito almeno una volta, poiché il controllo

16: Non tutti i linguaggi di programmazione le includono tutte e tre, in alcuni è stata fatta la scelta di averne una sola, solitamente o il **`while`** o il **`for`**.

Tabella 3.4.: Selezioni annidate e in cascata

Istruzioni C++	Diagrammi di flusso

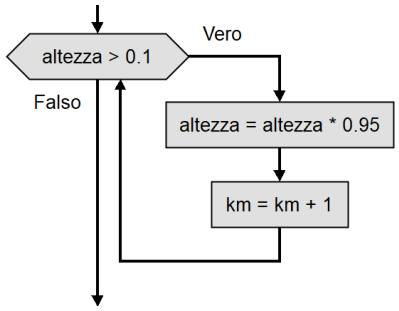
`if (a <= b && a <= c){`
`  std::cout << a << std::endl;`
`} else if (b <= a && b <= c){`
`  std::cout << b << std::endl;`
`} else {`
`  std::cout << c << std::endl;`
`}`
	

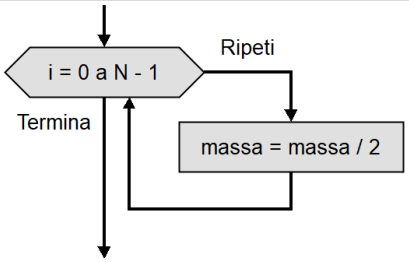
`if (a <= b && a <= c){`
`  std::cout << a << std::endl;`
`}`
`if (b <= a && b <= c){`
`  std::cout << b << std::endl;`
`}`
`if (c <= a && c <= b){`
`  std::cout << c << std::endl;`
`}`
	

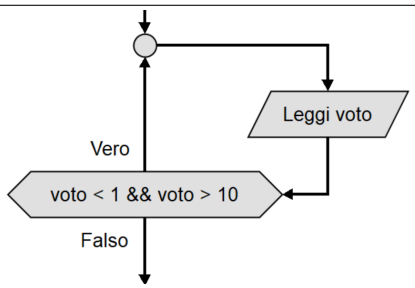
avviene in *coda*. Le situazioni in cui vale la pena utilizzarlo sono limitate, in quanto spesso si preferisce comunque il `while`.

Tabella 3.5.: I diversi tipi di iterazione in C++

Istruzioni C++	Diagrammi di flusso

`while (altezza > 0.1){`
`  altezza *= 0.95;`
`  km++;`
`}`
	

`for (int i = 0; i < N; ++i){`
`  massa /= 2;`
`}`
	

`do{`
`  std::cout << "Inserisci il voto: ";`
`  std::cin >> voto;`
`}while (voto < 1 || voto > 10);`
	

3.4. Esempi di programmi completi

Verranno adesso mostrati un paio di esempi che implementano problemi di carattere matematico, per evidenziare sia l'utilizzo del linguaggio C++ in contesti complessi, che le motivazioni che conducono alla scelta di utilizzare certe strutture di controllo piuttosto che altre.

Esempio 1: calcolo della funzione esponenziale con lo sviluppo in serie di Taylor

In questo esempio si vedrà come, dato un qualsiasi valore reale, sarà possibile calcolare il valore della funzione*esponenziale* in quel punto, utilizzando un metodo chiamato sviluppo in serie di Taylor. Se termini come *funzione*, *esponenziale*, *sviluppo in serie* potrebbero non suonare familiari, non è necessario preoccuparsi: è uno scenario piuttosto comune che il programmatore non conosca o conosca poco il contesto, in questo caso la matematica, ma potrebbe essere la compravendita di azioni come la gestione della logistica novale. Ciò

in generale non è un problema, a patto che il programmatore possa avere una descrizione sufficiente del problema che gli permetta di scriverne l'algoritmo risolutivo. Tornando al problema, esiste una formula che, dato un valore $x \in \mathbb{R}$, permette di calcolare e^x attraverso una serie di operazioni elementari, come mostrato qua sotto

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad \text{per ogni } x \in \mathbb{R}$$

Questa formula dice che si può ottenere e^x sommando tra di loro infiniti termini, ognuno dei quali è una frazione al cui numeratore si trova x^n e al denominatore il fattoriale di n. Siccome un algoritmo non può essere composto da infiniti passi, si può pensare di fermare questa somma ai primi N termini, avendo così una versione approssimata, ma comunque utile, del valore dell'esponenziale. Se ad esempio si decidesse di fermarsi ai primi 4 termini, cioè con N = 4, si avrebbe

$$e^x = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!}$$

L'input di questo problema sarà quindi il valore x e il numero di termini a cui si intende troncare la somma, che verrà indicato con N e come output un numero che è il valore dell'esponenziale che si vuole calcolare.

Una volta individuati l'input e l'output, dovrebbe essere semplice che ci si trova in presenza di un problema che per essere risolto necessita di un'iterazione, in quanto il risultato viene descritto in termini di somme ripetute. Inoltre il numero di termini della somma è individuato dalla variabile N, quindi il ciclo è *determinato* e il modo più adatto per implementarlo è utilizzando un **`for`**. Siccome poi i termini devono accumularsi per produrre il risultato, è necessario avere una variabile che faccia da accumulatore e altre variabili che, ad ogni giro, contengano il valore opportuno della potenza intera e del fattoriale. Individuati gli «ingredienti» fondamentali della soluzione, si può iniziare a scrivere il programma, che completato sarà qualcosa di simile al listato 3.8.

Listing 3.8: Calcolo dell'esponenziale con la serie di Taylor

```
#include <iostream>

int main() {
    int N, fattoriale = 1;
    double x, potenza = 1, risultato = 0;
    std::cout << "Inserisci il valore di x: ";
    std::cin >> x;
    std::cout << "Inserisci il numero di termini della
    somma: ";
```

```
9     std::cin >> N;
10    for (int i = 0; i < N; ++i) {
11        double termine = potenza / fattoriale;
12        risultato += termine;
13        fattoriale *= i + 1;
14        potenza *= x;
15    }
16    std::cout << "Il valore dell'esponenziale in " << x
17    << " è " << risultato << std::endl;
18    return 0;
19 }
```

Le variabili, definite nelle prime due righe del programma, rappresentano l'input `x` e `N`, la variabile `risultato` che conterrà appunto il risultato della computazione, e infine due variabili d'appoggio, `fattoriale` e `potenza`, che conterranno rispettivamente i valori di $n!$ e x^n.

Il ciclo `for` farà il numero di cicli inseriti dall'utente e, ad ogni giro, verranno aggiornati i valori di `fattoriale` e `potenza`, in questo modo:

- il fattoriale, essendo il prodotto dei primi n numeri naturali, ad ogni giro verrà moltiplicato con il valore di `i+1`, che quindi partirà da 1, successivamente sarà 2, e così via (riga 13)
- per la potenza il procedimento è simile, solo che, essendo la potenza per esponente naturale nient'altro che una serie di moltiplicazioni della base per se stessa, ad ogni giro la variabile verrà moltiplicata per la base `x` (riga 14)

Ogni singolo termine verrà quindi calcolato come rapporto tra i due valori appena discussi e verrà accumulato nella variabile `risultato`, che, all'uscita dal ciclo, conterrà al suo interno il valore di e^x. Per verificare il funzionamento del programma si calcoli l'esponenziale per un qualsiasi valore di x utilizzando 12 termini[17] e confrontando il risultato ottenuto con quello che produce una calcolatrice scientifica.

17: Si ricorda che il fattoriale cresce molto velocemente e già 13! non è più rappresentabile in maniera corretta come `int`.

Esempio 2: calcolo della radice quadrata con il metodo di bisezione

La radice quadrata è un'operazione che compare come passaggio intermedio in molte formule e quindi esiste già nel linguaggio C++ una funzione, `sqrt`, che calcola il valore della radice di un qualunque numero positivo.

Dovrebbe però essere chiaro che questo calcolo viene fatto attraverso una serie di passaggi intermedi composti solo da operazioni elementari. Il metodo che verrà affrontato in questo paragrafo è chiamato *metodo di bisezione* e il motivo sarà presto chiaro.

Volendo calcolare la radice di un numero positivo $N \geq 0$ si inizia individuando un intervallo $[a, b]$ all'interno del quale la radice sarà sicuramente compresa. Risulta piuttosto evidente che una scelta che garantisce questa condizione[18] è quella in cui $a = 0$ e $b = N$, poichè la radice è sicuramente minore di N, e, essendo il numero $N \geq 0$ anche la radice sarà maggiore o uguale a 0.

18: In realtà se $0 \leq N < 1$ allora l'intervallo iniziale sarà $[N, 1]$, per il resto il procedimento sarà lo stesso.

A questo punto l'idea è quella di trovare un stima della radice quadrata prendendo la media dell'intervallo $[a, b]$ come valore approssimato della radice quadrata. Chiamato m questo valore, ci possono essere solo due casi:

- m è una *sottostima* della radice quadrata, cioè $m < \sqrt{N}$, allora la radice si troverà nell'intervallo $[m, b]$
- m è una *sovrastima* della radice quadrata, cioè $m > \sqrt{N}$, allora la radice si troverà nell'intervallo $[a, m]$

Riapplicando lo stesso procedimento a intervalli che ogni volta sono la metà dell'intervallo precedente, dovrebbe essere chiaro che il valore m di stima della radice si avvicinerà sempre di più al valore $\sqrt{N}$. Il processo terminerà quando a e b saranno vicini «a sufficienza», dove «a sufficienza» viene stabilito dal programmatore a seconda della precisione che si vuole ottenere: se infatti $b - a < \epsilon$ allora $\left|m - \sqrt{N}\right| < \frac{\epsilon}{2}$: quindi scegliendo ad esempio la distanza ϵ tra a e b uguale a 0.001, allora l'errore sarà al massimo di 0.0005.

Ma come è possibile stabilire se il valore di m, che viene ricalcolato ogni volta, sia maggiore o minore della radice, dal momento che la radice non è conosciuta ed è proprio il punto finale a cui si vuole arrivare?

Difatti non è possibile. Però è possibile elevare al quadrato ambedue i lati della disequazione, poiché la relazione di minore o maggiore viene mantenuta. Quindi la condizione di verifica, ad esempio per il maggiore, indicata nei punti precedenti sarà:

$$m \cdot m > N \Rightarrow m > \sqrt{N}$$

che invece è verificabile perché sono conosciuti entrambi i lati della disequazione.

Per chiarire il procedimento si guardi la Figura 3.3, in cui si è preso come esempio N = 18. Al primo passaggio m varrà 9, che è una sovrastima della radice poiché $9 \cdot 9 > N$. Quindi il nuovo intervallo sarà $[0, 9]$, poiché può essere escluso che la radice si trovi nell'intervallo $[9, 18]$. Al secondo passaggio il nuovo valore sarà 4.5, che è nuovamente una sovrastima della radice, in quanto $4.5 \cdot 4.5 > 18$. Il nuovo intervallo diventerà $[0, 4.5]$ e al terzo passaggio m diventerà uguale a 2.25, che invece è una sottostima

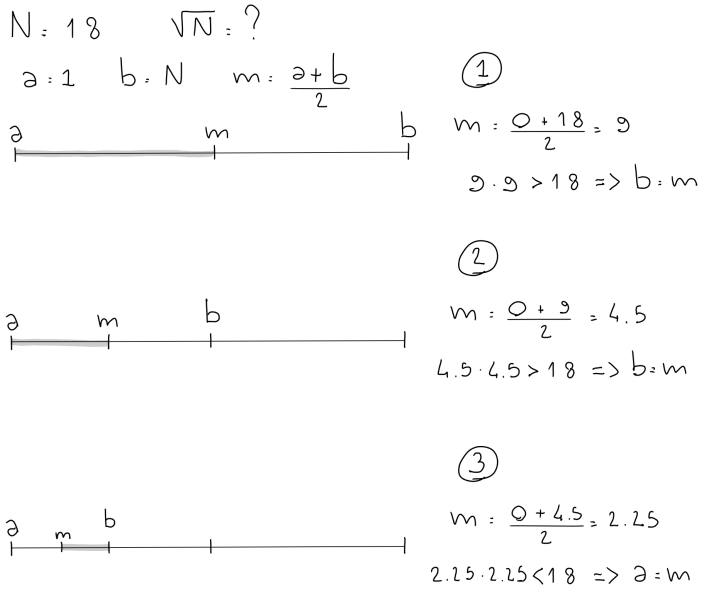


Figura 3.3.: Esempio di esecuzione dell'algoritmo di bisezione.

della radice e quindi stavolta sarà *a* che si «sposterà», rendendo il nuovo intervallo uguale a [2.25, 4.5].

Siccome ad ogni passaggio la lunghezza dell'intervallo dimezza, non ne serviranno molti per arrivare a un valore sufficientemente preciso.

Una volta compreso l'algoritmo non dovrebbe essere difficile implementarlo in linguaggio C++. Nella fase di ideazione, si dovrebbe notare che:

- siccome bisogna ripetere un procedimento varie volte, sarà necessario un costrutto iterativo.
- non conoscendo il numero di cicli da effettuare, ma la condizione di terminazione (intervallo sufficientemente piccolo), il costrutto `while` risulta più adatto del costrutto `for`
- dovendo poi controllare se *m* è una sottostima o una sovrastima bisognerà fare anche una selezione, quindi comparirà un `if-else`

Nel listato 3.9 si può vedere che, dopo aver fatto inserire all'utente il valore N di cui si vuole calcolare la radice, viene fatto un controllo per stabilire come impostare l'intervallo di partenza, secondo quanto indicato nella nota laterale.

Successivamente compare il `while`, la cui condizione può essere tradotta come «Continua a iterare fino a quando la distanza tra *a* e *b* è sopra l'errore che è stato impostato». All'interno del ciclo viene quindi calcolato il punto medio `m` e viene utilizzata la condizione di verifica per capire se si tratta di una sottostima o di una sovrastima, impostando di conseguenza il nuovo valore di *a* o di *b*.

In uscita dal ciclo m sarà una stima buona a piacere di $\sqrt{N}$. Si può anche notare che alla riga 1 è stata definita una variabile costante: con il valore impostato la radice sarà corretta fino alla quinta cifra decimale, se si volesse avere un'approssimazione migliore basterebbe rendere il valore di `EPSILON` ancora più piccolo.

Listing 3.9: Calcolo della radice quadrata con il metodo di bisezione

```
1  const double EPSILON = 0.00001;
2
3  int main()
4  {
5      double N;
6      double a, b, m;
7      std::cout << "Inserisci un numero positivo: ";
8      std::cin >> N;
9      if (N >= 1){
10         a = 0;
11         b = N;
12     } else {
13         a = N;
14         b = 1;
15     }
16     while(b - a > EPSILON)
17     {
18         m = (a + b)/2;
19         if (m*m > N){
20             b = m;
21         } else {
22             a = m;
23         }
24     }
25     std::cout << "La radice è " << m << std::endl;
26     return 0;
27 }
```

3.5. Esercizi

Studi

Esercizi elementari per prendere confidenza con le varie istruzioni presentate nel capitolo.

Istruzioni di I/O

In questa serie di esercizi viene richiesto di usare solo le istruzioni di I/O e le operazioni aritmetiche fondamentali.

1. Chiedere all'utente di inserire una variabile intera e poi stamparla
2. Chiedere all'utente di inserire una variabile float e poi stamparla
3. Chiedere all'utente di inserire una variabile double e poi stamparla
4. Chiedere all'utente di inserire una variabile intera, una float e una double e poi stamparle in ordine inverso di acquisizione
5. Chiedere all'utente di inserire una variabile intera e poi stampare la variabile aumentata di 10 unità, di 100 unità e di 1000 unità
6. Chiedere all'utente di inserire una variabile intera e poi stampare la variabile moltiplicata per 100, divisa per 10 (divisione intera) e il resto della divisione della variabile per 3
7. Chiedere all'utente di inserire una variabile float e stamparla dopo averla divisa per un miliardo
8. Chiedere all'utente di inserire una variabile double e stamparla dopo averla divisa per mille miliardi
9. Chiedere all'utente di inserire una variabile float e una intera e stampare il prodotto tra le due
10. Chiedere all'utente di inserire due variabili intere di valore compreso tra un miliardo e due miliardi e stamparle, successivamente stamparne la somma, la differenza e il prodotto, interpretando i risultati ottenuti
11. Chiedere all'utente di inserire due variabili double di valore compreso tra un miliardo e due miliardi e stamparle, successivamente stamparne la somma, la differenza e il prodotto, interpretando i risultati ottenuti

Selezione

In questa serie di esercizi viene richiesto di usare solo le istruzioni di I/O e i costrutti di selezione `if`, `if-else`

1. Chiedere all'utente di inserire due variabili intere e stampare la maggiore tra le due
2. Chiedere all'utente di inserire due variabili intere e stampare quelle con valore positivo (tutte e due, solo una o nessuna)
3. Chiedere all'utente di inserire due variabili n e m e verificare se n è almeno il doppio di m.
4. Chiedere all'utente di inserire tre numeri float e verificare se il primo e il secondo hanno la stessa «distanza» fra di loro che il secondo e il terzo
5. Chiedere all'utente di inserire due numeri float, dove il primo rappresenta un prezzo e il secondo un sconto in percentuale (ad esempio 10%) e stampare il prezzo normale o scontato, poichè lo sconto si applica solo agli articoli con prezzo superiore ai 50 euro.

6. Chiedere all'utente di inserire un numero intero e un float e calcolare somma, differenza, prodotto e divisione, stampando ogni singolo risultato solo se è compreso tra 100 e 1000
7. Chiedere all'utente di inserire due numeri double, moltiplicarli tra loro e verificare se il prodotto è inferiore a 10 volte la loro somma
8. Chiedere all'utente di inserire 4 numeri interi e verificare quale di essi è il maggiore e quale il minore
9. Chiedere all'utente di inserire tre numeri a, b e c che fanno parte della proporzione $a:b = c:\mathbf{x}$ e calcolare il valore di **x**, stamparlo e comunicare se è maggiore, minore o uguale a 100.
10. Chiedere all'utente di inserire un numero compreso tra 1 e 12 che rappresenta un mese (1-gennaio, 2-febbraio ecc.) e stampare il numero di giorni che contiene quel mese, ignorando le regole dell'anno bisestile.

Iterazione

In questa serie di esercizi viene richiesto di usare solo le istruzioni di I/O, i costrutti di selezione `if`, `if-else` e quelli di iterazione `for`, `while`, `do-while`.

1. Chiedere all'utente di inserire una serie di numeri interi e terminare l'inserimento al primo numero negativo inserito. Stampare quanti numeri sono stati inseriti
2. Chiedere all'utente di inserire due numeri positivi, facendoli reinserire se sono negativi, in modo che alla fine risultino entrambi positivi. Mostrare poi la loro somma.
3. Chiedere all'utente di inserire un numero N positivo e stampare i quadrati di tutti i numeri compresi tra 1 e N
4. Chiedere all'utente di inserire un numero N positivo e stampare i cubi di tutti i numeri compresi tra N e 1, in quest'ordine
5. Chiedere all'utente di inserire due numeri N e M positivi e stampare i quadrati di tutti i numeri compresi tra il minore e il maggiore dei due
6. Chiedere all'utente di inserire due numeri N e M positivi e stampare i quadrati di tutti i numeri compresi tra il minore e il maggiore dei due, solo se sono pari
7. Chiedere all'utente di inserire due numeri N e M positivi e stampare il prodotto di tutti i numeri compresi tra il minore e il maggiore dei due
8. Chiedere all'utente di inserire un numero N positivo e stampare tutti i divisori di N
9. Chiedere all'utente di inserire un numero N positivo e verificare quanti numeri compresi tra 1 e N sono multipli di 5
10. Chiedere all'utente di inserire un numero N positivo e verificare quanti numeri compresi tra 1 e N sono multipli sia di 3 e che di 5
11. Chiedere all'utente di inserire un numero N positivo e un carattere C e stampare una serie composta dal carattere C ripetuto N volte
12. Chiedere all'utente di inserire un numero N positivo e un carattere C e stampare una serie composta dal carattere C ripetuto N volte, ma andando a capo ogni volta che vengono stampati 5 caratteri C
13. Chiedere all'utente di inserire due numeri N e M positivi e stampare N righe composte da M caratteri 'x' ognuna
14. Chiedere all'utente di inserire un numero N positivo e far inserire N numeri float e stampare la media
15. Chiedere all'utente di inserire un numero N positivo e far inserire N numeri float e stampare la media solo dei numeri maggiori di 100

16. Chiedere all'utente di inserire un numero N positivo e due numeri float A e B. Successivamente far inserire N numeri float e contare quanti sono i valori inseriti compresi tra A e B

Esercizi

1. Un broker quando esegue delle transazioni (ad esempio acquisto/vendita di azioni), riceve una commissione in base all'importo della transazione effettuata secondo la seguente tabella:

Transazione	Commissione
Fino a 2500 €	30 € + 1.7%
Da 2500 € fino a 6250	56 € + 0.66%
Da 6250 € fino a 20000 €	76 € + 0.34%
Oltre i 20000 €	100 € + 0.22%

   Inoltre la tariffa minima è di 39 €.
   Si scriva un programma che dato l'importo di una transazione calcoli la commissione dovuta al broker.
2. Si scriva un programma che, dopo aver chiesto all'utente se vuole convertire una temperatura da Celsius a Fahrenheit o viceversa, richieda di immettere una temperatura ed esegua la conversione richiesta.
   Si ricorda che la conversione da Celsius a Fahrenheit si effettua con la seguente formula:

$$F^\circ = C^\circ \times \frac{9}{5} + 32^\circ$$

3. Dato un orario hh:mm nella forma a 24H, trasformarlo nell'equivalente a 12H con AM/PM (attenzione che 12:00 PM sono mezzogiorno e 12:00 AM sono mezzanotte).
4. Si scriva un programma che calcoli e stampi i valori di energia cinetica e energia potenziale in funzione del tempo di un punto materiale di massa m soggetto alla forza di gravità mg.
   Si ricorda che l'energia cinetica si calcola con la formula $T = \frac{1}{2}mv^2$ e quella potenziale con $U = mgh$. Come input devono essere forniti i valori di massa m, velocità iniziale v_0, altezza iniziale h_0 e tempo t.
   Si ricorda inoltre che in presenza di accelerazione **g**, la velocità vale $v(t) = v_0 - gt$ e l'altezza del punto vale $h(t) = h_0 + v_0 t - \frac{1}{2}gt^2$.
   Si controlli inoltre che, dati i valori di tempo e altezza iniziale, il punto non si trovi in una posizione negativa rispetto all'origine del sistema di riferimento. Nel caso ciò si verifichi, il programma deve stampare in che momento il punto ha toccato terra e con quale energia cinetica.
5. Scrivere un programma che inserendo il proprio giorno e mese di nascita, comunichi il segno zodiacale corrispondente alla data inserita.
6. Scrivere un programma che traduca un numero compreso tra 1 e 100 inserito dall'utente nel corrispondente numero romano.
7. Scrivere un programma che faccia inserire all'utente due numeri interi compresi tra 1 e 12 che rappresentano due mesi (1 gennaio, 2 febbraio ecc.) e dia come output il numero di giorni che passano dall'inizio del primo mese inserito alla fine del secondo inserito.
8. Data la formula $F = m \times a$ (F = Forza, m = massa, a = accelerazione) scrivere un programma che chieda all'utente di inserire due parametri a sua scelta (tra i tre che compongono la formula) e calcoli il valore del terzo. Ad esempio se l'utente avesse scelto di inserire forza = 13 e accelerazione = 2 il programma dovrebbe stampare

```
Forza = 13
Accelerazione = 2
```

```
Il risultato è Massa = 6.5
```

9. Scrivere un programma che dica se un anno inserito dall'utente è bisestile o no. Un anno è bisestile* se è un multiplo di 4, ad esempio il 1996. Tra questi però ci sono anni che fanno eccezione e sono i multipli di 100: ad esempio il 1900 non è bisestile pur essendo multiplo di 4. Infine c'è un'ultima regola che dice che i multipli di 400 sono bisestili sempre: ad esempio il 2000 era un bisestile, pur essendo un multiplo di 100.
10. Scrivere un programma che legga da tastiera una sequenza di numeri interi positivi. Il programma, a partire dal primo numero introdotto, deve stampare ogni volta la media di tutti i numeri introdotti fino a quel momento. Il programma terminerà quando il numero inserito sarà negativo, senza utilizzarlo per stampare la media.
11. Si scriva un programma che, dati N numeri interi, con N anch'esso inserito dall'utente, stampi qual è il valore massimo e il valore minimo tra tutti quelli inseriti.
12. Una ditta deve calcolare le paghe settimanali dei suoi impiegati con il seguente criterio:
    - se le ore lavorate sono minori o uguali a 40 la paga è di 22 euro all'ora
    - tutte le ore sopra le 40 vengono pagate 32.5 euro all'ora.

    Scrivere un programma che permetta di inserire il numero di ore per ogni impiegato, stampi subito la paga settimanale di quell'impiegato, termini quando si inserisce un –1 e prima di terminare stampi il totale (la somma) di tutte le paghe settimanali e il numero di impiegati.
13. Scrivere un programma che simuli il gioco di pari e dispari. Fatto scegliere al computer un numero casuale compreso tra 0 e 5 (che non deve essere mostrato all'utente), il programma chiederà all'utente di scegliere tra pari (1) e dispari (2), successivamente di inserire un numero compreso tra 0 e 5 e dica chi ha vinto a pari e dispari, verificando se la somma dei due numeri sia pari o dispari. Il programma infine dovrà mostrare delle frasi come queste di esempio: «Bravo, hai vinto, il tuo 3 e il 3 del computer hanno somma pari uguale a 6 e tu hai scelto pari» oppure «Peccato, il computer ha tirato 5 e con il tuo 2 fa 7, purtroppo avevi scelto pari».
14. Si scriva un programma che calcoli dopo quanti rimbalzi una pallina di gomma, lanciata da un'altezza H in metri inserita in input dall'utente, può essere considerata ferma, considerando che ad ogni rimbalzo l'altezza che raggiunge è il 99% dell'altezza dalla quale è partita e che quando l'altezza di rimbalzo diventa inferiore a 1 millimetro può essere considerata ferma.
15. Si scriva un programma che dato in input un intero positivo N stampi un rombo utilizzando il simbolo *, come mostrato nei seguenti esempi, dove N è la lunghezza del lato.

* Le regole della bisestilità sono state introdotte per sincronizzare l'anno solare con l'anno astronomico ed evitare così che nel lungo periodo si abbia uno slittamento delle stagioni.

Esempio con N = 1

```
*
```

Esempio con N = 2

```
  *
 ***
*****
 ***
  *
```

Esempio con N = 3

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

16. Si scriva un programma che dato in input un mese (come numero compreso tra 1 e 12) e il giorno della settimana del primo di quel mese (come numero compreso tra 1 e 7), stampi un calendario mensile come nell'esempio seguente, in cui il mese è maggio (31 giorni) e il primo giorno del mese è 4 (un giovedì).

```
 L   M   M   G   V   S   D
             1   2   3   4
 5   6   7   8   9  10  11
12  13  14  15  16  17  18
19  20  21  22  23  24  25
26  27  28  29  30  31
```

17. Si scriva un programma che dato in input un intero positivo **N** disegni una «capsula» come negli esempi sotto riportati.

Esempio con N = 1

```
 _
/ \
|  |
\_/
```

Esempio con N = 2

```
  __
 /  \
/    \
|    |
|    |
\    /
 \__/
```

18. Scrivere un programma che stabilisca se un numero inserito dall'utente è divisibile utilizzando la seguente regola: un numero è divisibile per 11 se, indicato con **SP** la somma delle cifre di posto pari e con **SD** quella delle cifre di posto dispari, il valore assoluto della differenza, cioè $|SP - SD|$, vale 0. Se il valore di questa differenza fosse maggiore di 10, il procedimento va riapplicato fino a quando si ottiene 0, quindi il numero è divisibile per 11, oppure è un numero compreso tra 1 e 10, e quindi non è divisibile per 11. Verificare infine la correttezza di quanto fatto utilizzando l'operatore modulo (%), per vedere se si ottiene lo stesso risultato.
19. Scrivere un programma che chieda all'utente di indovinare un numero compreso tra 1 e 1000 scelto a caso dal computer. Il computer dovrà comunicare se il numero proposto dall'utente è più grande o più piccolo del numero da indovinare e continuare fino a quando l'utente non lo indovinerà, comunicando alla fine il numero di tentativi fatti prima di indovinare il numero. Si utilizzi la funzione di libreria `rand` per generare il numero casuale.
20. Scrivere un programma per il calcolo del minimo comune multiplo fra due numeri interi positivi.
21. Scrivere un programma per il calcolo del Massim Comun Divisore fra due numeri interi positivi.
22. Scrivere un programma che stampi le terne pitagoriche i cui componenti siano minori di 500. Una terna pitagorica è un insieme di tre numeri interi positivi (A, B, C) tali che $A^2 + B^2 = C^2$. Due terne pitagoriche sono considerate la stessa terna se la sono differenti solo per l'ordine dei loro elementi, quindi la terna (A, B, C) è la stessa terna che (B, C, A). Il programma non dovrà stampare più volte la stessa terna.
23. Scrivere un programma per la rappresentazione del triangolo di Floyd. Il triangolo di Floyd è un triangolo rettangolo che contiene numeri naturali, definito riempiendo le righe del triangolo con numeri consecutivi e partendo da 1 nell'angolo in alto a sinistra. Si consideri ad esempio il caso N=5. Il triangolo di Floyd sarà il seguente:

    ```
    1
    2 3
    4 5 6
    7 8 9 10
    11 12 13 14 15
    ```

    L'utente dovrà inserire il numero N e il programma dovrà stampare il triangolo di Floyd corrispondente.
24. Scrivere un programma che dati in input due valori interi positivi n e k calcoli il coefficiente binomiale utilizzando la seguente formula:

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \quad n, k \in \mathbb{N}; 0 \leq k \leq n$$

    Si ricorda che la notazione $n!$ si legge come *n fattoriale* e non è altro che il prodotto di tutti i numeri naturali compresi tra 1 e n.
25. Si scriva un programma che, dato in input un numero intero positivo N, lo stampi al contrario. Se ad esempio il numero fosse 1234 verrà stampato come 4321, 1200 come 21.
26. Si scriva un programma che dato in input un intero positivo N disegni un numero N di onde secondo il seguente schema:

Esempio con N = 1

```
 _
/ \
   \_/
```

Esempio con N = 2

```
 _     _
/ \   / \
   \_/   \_/
```

Esempio con N = 3

```
 _     _     _
/ \   / \   / \
   \_/   \_/   \_/
```

27. Utilizzando le formule che descrivono la traiettoria parabolica di un oggetto dato l'angolo α e la velocità iniziale v_0, stampare la posizione dell'oggetto in termini delle sue coordinate x e y, da quando parte a quando tocca terra, a intervalli di 0.1 secondi. Si supponga che il punto di lancio si trovi nelle coordinate (0,0) del sistema di riferimento, quindi $h_0 = 0$.

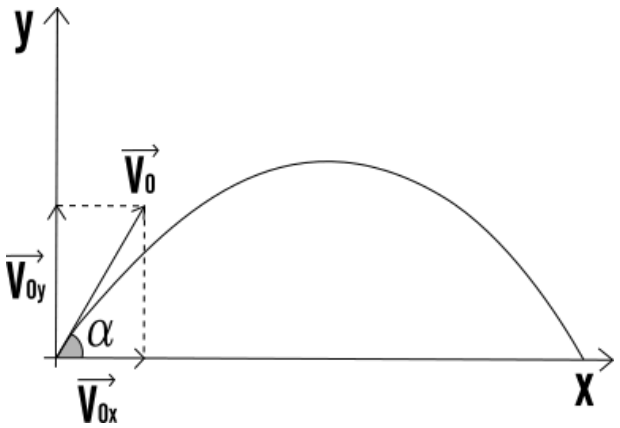


Figura 3.4.: Traiettoria parabolica di un oggetto.

Si ricorda che lungo l'asse parallelo al piano di lancio la velocità è costante, mentre lungo l'asse perpendicolare il moto è uniformemente accelerato con accelerazione $-g$, con $g = 9.81$. La formula del moto uniformemente accelerato è:

$$h(t) = h_0 + v_0 t - \frac{1}{2} g t^2, \text{ con } h(t) \text{ altezza del proiettile da terra all'istante } t$$

Si ricorda infine che la velocità iniziale v_0 deve essere scomposta nelle due componenti, parallela e perpendicolare al terreno, utilizzando le note relazioni che utilizzano il seno e il coseno, cioè:

$$v_{0x} = v_0 \cdot \cos(\alpha)$$
$$v_{0y} = v_0 \cdot \sin(\alpha)$$

28. Si supponga di avere un macchinario che, data una lastra di metallo di forma rettangolare con base b e altezza h, tagli un quadrato da ognuno degli spigoli, in modo che poi la si possa piegare e facendo combaciare i lati così ottenuti si possa ottenere un parallelepipedo, come si può vedere dalla figura Figura 3.5. A seconda della dimensione dei lati dei quadrati ritagliati (indicati in figura Figura 3.5 con 1, 2, 3 e 4) il volume risulta diverso. Il programma, dati in input i valori di base e altezza della lastra, dovrà trovare il lato del quadrato che permette di massimizzare il volume del parallelepipedo.

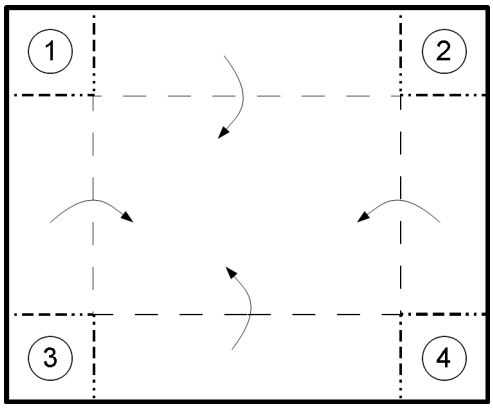


Figura 3.5.: Esempio di come piegare la lastra.

29. Calcolare il valore del numero e di Eulero, utilizzando la definizione tramite la sommatoria definita sotto, fermandosi ai primi N termini, con N inserito dall'utente.

$$e := \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

30. Si scriva un programma che calcoli il valore della funzione *coseno* utilizzando il seguente sviluppo in serie di Taylor:

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \quad \text{per ogni } x$$

fermando il calcolo della serie al termine N-esimo, con N e x inseriti dall'utente. In questa formula il valore di x deve essere in *radianti*.

Progetti

Il gioco del Craps

In alcuni spettacoli televisivi[†] in cui i protagonisti giocano al Casinò, viene mostrato un gioco in cui bisogna lanciare una coppia di dadi e, in base alla somma dei numeri usciti, può vincere il giocatore che ha lanciato i dadi o il banco[‡]. Spesso questo gioco è il gioco del Craps.

Scrivere un programma che permetta di giocare a questo gioco. Le regole del Craps sono le seguenti:

- Viene lanciata la coppia di dadi e se:
  - esce 7 o 11, il giocatore vince
  - esce 2,3 o 12, il giocatore perde
  - qualsiasi altro valore diventa il punteggio (**target**) per i tiri successivi
- quando il giocatore non vince o non perde al primo lancio, il gioco prosegue fino a quando:
  - esce il **target**, cioè il valore fatto al primo lancio, il giocatore vince
  - esce 7, il giocatore perde
  - esce qualsiasi altro valore, in questo caso si ritirano i dadi

  Si suggerisce di organizzare il programma in modo che il giocatore disponga di una somma di partenza da utilizzare durante le giocate, che aumenti o diminuisca in base alle vittorie e alle sconfitte. Per simulare il lancio dei dati si utilizzi la funzione `rand`. Per l'interazione con l'utente, si faccia in modo che il lancio dei dadi sia scatenato dall'inserimento del numero 1 da parte dell'utente, poichè la gestione dell'input in *console* non permette nativamente di intercettare la semplice pressione di un tasto, ma richiede sempre l'inserimento di qualcosa seguito dalla pressione del tasto Invio[§].

Punteggi del tennis

Nelle partire di tennis, in ogni gioco (game) l'arbitro deve assegnare il punto a chi vince lo scambio, solo che il sistema dei punteggi é piuttosto macchinoso:

- i primi tre punti di ogni giocatore vengono denominati 15, 30 e 40, rispettivamente
- nel caso un giocatore sia a 40 e l'altro no, se fa il punto vince il game
- se il punteggio arriva a 40 per entrambi, il primo giocatore che fa il punto successivo acquisisce un vantaggio: se quello stesso giocatore fa un altro punto allora vince il game, altrimenti si torna a 40 pari e si prosegue così finchè uno dei due vince il game, con le regole appena enunciate.

Scrivere un programma per la gestione di un game di tennis, dove l'utente è l'arbitro e ad ogni punto giocato inserisce un 1 o un 2, a seconda che il punto lo vinca il giocatore 1 o il giocatore 2. Per ogni punto inserito il programma deve mostrare il punteggio attuale e, quando il game finisce, scrivere chi ha vinto. Una possibile sequenza di esecuzione per il prestigioso torneo Roland Garros è la seguente:

[†] Ad esempio nell'episodio «The One in Vegas» della serie Friends, Monica e Chandler giocano proprio a questo gioco.
[‡] Spoiler: nel lungo periodo il banco vince sempre, come in tutti i giochi d'azzardo.
[§] Esistono dei «trucchi» per leggere la pressione di un tasto, ma in generale sono dipendenti dall'ambiente in cui il programma gira e ne è scoraggiato l'utilizzo.

```
Roland Garros Game Manager
0 - 0
Chi ha vinto il punto (1 o 2): 1
15 - 0
Chi ha vinto il punto (1 o 2): 2
15 - 15
Chi ha vinto il punto (1 o 2): 2
15 - 30
Chi ha vinto il punto (1 o 2): 2
15 - 40
Chi ha vinto il punto (1 o 2): 1
30 - 40
Chi ha vinto il punto (1 o 2): 1
40 - 40
Chi ha vinto il punto (1 o 2): 1
Vantaggio giocatore 1.
Chi ha vinto il punto (1 o 2): 2
Vantaggio pari.
Chi ha vinto il punto (1 o 2): 2
Vantaggio giocatore 2.
Chi ha vinto il punto (1 o 2): 2
Fine del game, ha vinto il giocatore 2
```

Funzioni 4.

Introduzione

Dopo aver visto come scrivere algoritmi in C++ per la soluzione di semplici problemi, ci si potrà già essere resi conto che più un programma diventa lungo, più complessa diventa sia la fase di scrittura che quella, eventuale, di una sua rilettura e modifica in futuro. Sono già state messe in atto tutta una serie di misure per rendere il codice più semplice da leggere e scrivere, alcune a livello dell'IDE, come ad esempio il *syntax hilighting*, altre a livello di organizzazione del codice, come l'indentazione o la scelta di nomi significativi di variabili, ma la crescita delle dimensioni e della complessità di un programma richiedono l'utilizzo di strumenti più *potenti*.

Le *funzioni*, già incontrate in alcuni programmi, sono, storicamente, una delle prime risposte alla crescita della complessità dei programmi: si può dire che, mentre *variabili*, *operatori matematici e logici*, *strutture di controllo* sono una necessità per poter scrivere dei programmi, le funzioni assolvono invece un compito di tipo «organizzativo» e quindi potrebbero anche non esserci, ma nella pratica è impossibile farne a meno.

4.1. Funzioni

Il concetto di *funzione* è simile a come viene inteso in matematica o in altri campi scientifici,con alcune differenze che verranno analizzate nel seguito del paragrafo.

L'idea di base è quella di raggruppare in un solo punto il codice necessario per eseguire un compito specifico (trovare la radice quadrata di un numero, fare il totale di una fattura, stampare un biglietto aereo, ecc.) e isolarlo dal resto del programma, in modo da ottenere una suddivisione dei compiti tra pezzi di codice «specializzati», utilizzabili in qualunque punto del programma. I principali vantaggi delle funzioni sono quindi:

- ogni funzione può essere sviluppata indipendentemente dal resto del codice e testata separatamente
- posso riutilizzare algoritmi comuni attraverso la creazione di librerie[1] di funzioni (che di fatto è quello che avviene nella libreria standard del C++ o nell'utilizzo di librerie esterne, come la libreria grafica RayLib che si trova in appendice)

1: In gergo informatico per *libreria* si intende una collezione di funzionalità, generalmente coerenti e relative a uno scopo preciso, sviluppate per essere usate a supporto di altri programmi. Il termine stesso deriva da un'erronea traduzione della parola inglese *library*, ormai entrato nell'uso comune dei programmatori italiani.

- il programma nel suo complesso diventa più facile da sviluppare e mantenere, poiché non è più una lunga sequenza di istruzioni di basso livello, ma l'insieme di chiamate di funzioni che svolgono operazioni a un livello di astrazione più elevato.

Definizione di funzione

Una funzione è un insieme coerente di istruzioni, che servono a svolgere un compito di carattere generale, e che restituisce un'informazione o svolge un'azione, o entrambe le cose, a fronte di informazioni che le vengono fornite attraverso una lista di *parametri*.

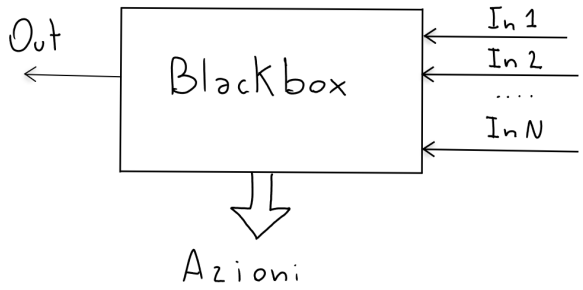


Figura 4.1.: Rappresentazione generale di una funzione

Una rappresentazione grafica del concetto di funzione può essere vista in Figura 4.1, dove, al contrario di come spesso viene disegnata in altri contesti, la funzione è una «blackbox»[2] che riceve degli input $\text{In}_1, \text{In}_2, \ldots, \text{In}_n$ a destra e fornisce in uscita un output Out, mostrato a sinistra, ed eventualmente effettua delle azioni che modificano lo stato del sistema (ad esempio stampa a schermo, apre una finestra, instaura una connessione di rete, ...).

2: Con il termine *blackbox* o *scatola nera* ci riferisce ad un sistema di cui si conoscono gli input e gli output, ma di cui è sconosciuto il meccanismo per la produzione dei secondi a partire dai primi.

La scelta di porre gli input a destra e gli output a sinistra deriva dall'analogia con la sintassi effettiva delle funzioni nei linguaggi C/C++ (e in molti altri), in cui gli input, chiamati *parametri*, si trovano a destra del nome della funzione e l'output, chiamato *valore di ritorno*, si trova a sinistra del nome di funzione. Quella rappresentata è la forma più generale di funzione, può essere che alcune funzioni non abbiano parametri o valore di ritorno o manchino di entrambi.

Utilizzo di funzioni

Nel trattare le funzioni ci si può trovare fondamentalmente in due situazioni, in parte sovrapposte:

1. come utilizzatore di funzioni scritte da altri: in questo caso non è necessario sapere *come funzionano*, cioè quali sono gli

algoritmi che internamente permettono loro di fare ciò che fanno, basta sapere *come utilizzarle*. Questa è la situazione a *blackbox*, in cui ci si appoggia sul lavoro fatto da altri programmatori per velocizzare la scrittura del proprio codice, renderlo più robusto e modulare. Tutto questo richiede che le funzioni siano corrette e che ci si fidi del fatto che lo siano
2. come progettisti di funzioni scritte in proprio: in questo caso invece tutta la responsabilità di implementare una funzione ricade sul programmatore, che poi, una volta progettata e realizzata, si sposterà nella situazione precedente, nella quale utilizzerà quanto scritto.

Per quanto riguarda il primo punto, il programmatore si troverà a *chiamare*[3] funzioni scritte da altri (o anche da se stesso), invece per il punto 2, il programmatore dovrà *definire* le proprie funzioni. Nel prossimo paragrafo si vedrà come definire una funzione, in quanto è l'attività da cui «nascono» le funzioni.

3: Il termine *chiamata di funzione* è un termine tecnico che deriva dal nome dell'istruzione *assembly* che permette di avviare una funzione, che è proprio `call`.

4.2. Definizione di funzione

Sintatticamente una funzione viene definita nel seguente modo:

```
valore_di_ritorno nome_funzione(lista_dei_parametri)
```

dove:

- **`valore_di_ritorno`**: può essere *void*, se non ha necessità di ritornare nessuna informazione al *chiamante*, oppure un qualsiasi tipo di dato elementare (**`int`**. **`char`**, **`float`**, ecc.) o definito dall'utente (struttura o classe, come si vedrà in seguito)
- **`nome_funzione`**: è un qualsiasi identificatore valido (come i nomi di variabili) che indica la semantica della funzione, cioè il suo scopo
- **`lista_parametri`**: una lista di variabili, elencate ognuna con tipo e nome, separate da virgola.

L'insieme del valore di ritorno, del nome e della lista di parametri è anche definito *firma*, *signature* in inglese, perché indica, a chi dovrà usare la funzione, tutte le informazioni necessarie per poterla utilizzare all'interno del proprio codice. Inoltre la firma è anche necessaria al compilatore per verificare che la chiamata della funzione corrisponda alla sua definizione, oltre che per il nome, anche per la lista dei parametri e il valore di ritorno. Se, ad esempio, il numero dei parametri nella firma fosse 2 e, nel punto di chiamata, si passasse solo 1 parametro, il compilatore segnalerebbe l'errore e impedirebbe di completare la compilazione.

Per comprendere meglio il significato di funzione ricorriamo a un esempio semplicissimo: supponiamo di voler sapere quale sia il maggiore tra due numeri interi passati come parametri. Un codice possibile[4] per raggiungere lo scopo è quello presentato di seguito:

```
1 int maggiore(int r, int s){
2     int max;
3     if (r > s)
4         max = r;
5     else
6         max = s;
7     return max;
8 }
```

4: Il codice è volutamente didattico per evidenziare alcuni aspetti legati al passaggio di parametri, un'implementazione più realistica utilizzerebbe una sola riga di codice.

Listing 4.1: Esempio di definizione di una funzione.

In Figura 4.2 si possono vedere a cosa corrispondano il nome, i parametri e il valore di ritorno in questa funzione di esempio.

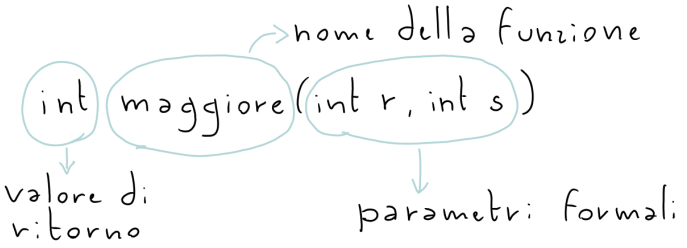


Figura 4.2.: Firma di una funzione

Per concludere, in Figura 4.3 si può vedere una metafora di una funzione, che rappresenta il processo per la produzione di pane. In questo esempio chi necessita di pane, il chiamante, può portare al forno, che rappresenta la funzione, i parametri attuali, che saranno una certa quantità di farina, sale e lievito. Tramite un processo (algoritmo) non necessariamente conosciuto dal chiamante, all'interno del forno verranno opportunamente mescolati, impastati e cotti gli ingredienti per restituire una quantità di pane dipendente dai parametri forniti (il valore di ritorno).

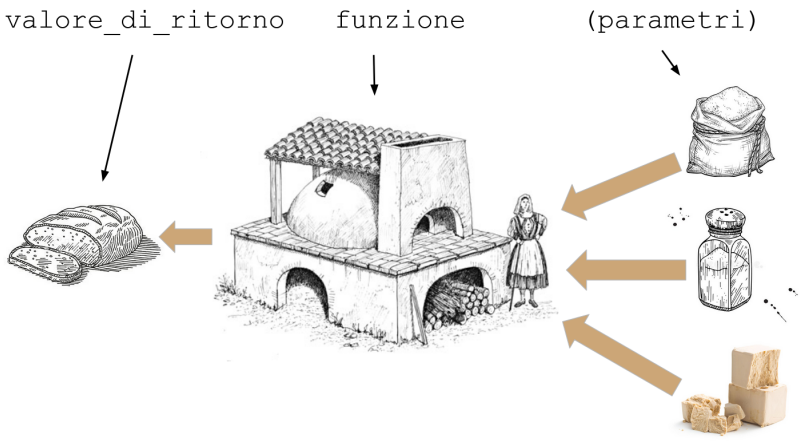


Figura 4.3.: Metafora di una funzione.

4.3. Chiamata di funzione

Finora abbiamo solo definito una funzione, ma la sua utilità risiede nella possibilità di chiamarla dove necessario, come già indicato nel paragrafo precedente. Al momento della chiamata, come si può vedere nel codice 4.2, nella lista dei parametri vengono inseriti dei valori, costanti o contenuti in variabili, che saranno oggetto dell'elaborazione che avverrà all'interno della funzione.

Listing 4.2: Esempio di chiamata di una funzione.

```
int main(){
    int a, b, c;
    std::cout << "Inserisci due valori interi: ";
    std::cin >> a >> b;
    c = maggiore(a, b);
    std::cout << "Il maggiore tra " << a << " e "
        << b << " è " << c << std::endl;
    return 0;
}
```

I parametri possono essere meglio specificati indicandoli nei seguenti due modi:

Parametri formali

I parametri formali sono quelli che compaiono all'interno della firma della funzione quando viene definita e servono per indicare, nel corpo della stessa, quali saranno le variabili su cui lavorare e come trattarle.

Parametri attuali

I parametri attuali sono quelli che compaiono tra le parentesi tonde al momento della chiamata, possono essere costanti o variabili, e i loro valori verranno copiati nei parametri formali per effettuare l'elaborazione specificata nel corpo della funzione.

Il nome dei parametri formali è irrilevante, in quanto la copia dei valori avverrà rispetto alla posizione, nel senso che il primo valore contenuto in un parametro attuale verrà copiato nel primo parametro formale, il secondo nel secondo e così via, come bene si può vedere in Figura 4.4.

Nel programma principale, all'interno del `main`, compaiono tre variabili locali, `a`, `b` e `c`, e i valori delle prime due vengono utilizzati nella chiamata di funzione (passaggio 1 in figura). Questo fa si che i valori contenuti in `a` e `b` vengano copiati all'interno dei parametri formali `r` e `s`, che sono a tutti gli effetti delle variabili contenute «all'interno» della funzione `maggiore`. Detta funzione esegue quindi le operazioni sui valori passati e inserisce il risultato all'interno

della variabile `max` (si veda il codice 4.1), che viene poi «ritornata» e assegnata alla variabile `c` (passaggio 2 in figura).

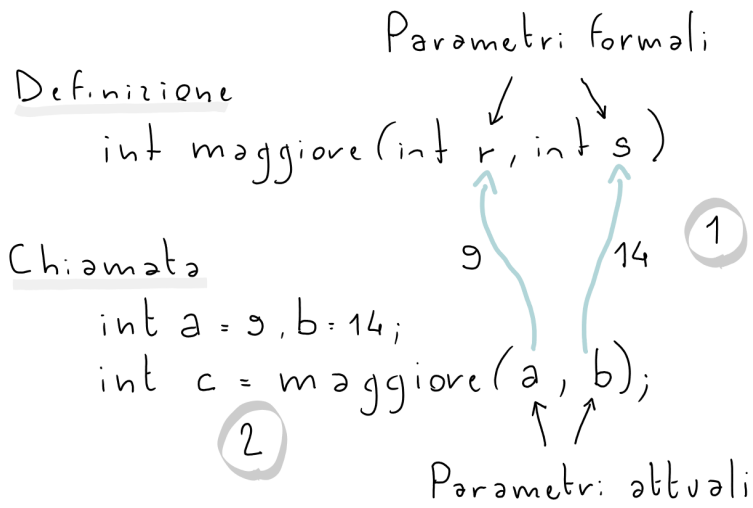


Figura 4.4.: Passaggio di parametri per copia.

In che punto di un programma può essere chiamata una funzione? La regola generale è che una funzione può essere chiamata laddove il corrispondente tipo del valore di ritorno sia utilizzabile. Può essere quindi utile pensare a una funzione come a una variabile di tipo corrispondente al proprio valore di ritorno, in modo da poter sempre porsi la domanda: «È possibile usare una variabile di tipo X in questa posizione?». Se la risposta è affermativa allora anche una funzione con valore di ritorno di tipo X può essere utilizzata, se invece la risposta è negativa allora nemmeno la funzione può essere utilizzata.

Come si è già avuto modo di dire, non tutte le funzioni hanno necessità di ritornare un valore: ci sono dei casi, a volte creati a scopo didattico, in cui la funzione viene pensata per avere degli effetti piuttosto che per fare delle elaborazioni che producano un qualche tipo di risultato.

Un esempio semplice può essere visto nel codice 4.3

Listing 4.3: Funzione per la stampa di un quadrato di asterischi.

```
1  void stampa_quadrato(int lato){
2      for (int i = 0; i < lato; ++i){
3          for (int j = 0; j < lato; ++j){
4              std::cout << "*";
5          }
6          std::cout << std::endl;
7      }
8  }
```

In questo esempio didattico, la funzione non ritornerà niente nel contesto in cui è stata chiamata, come si può chiaramente vedere dal fatto che il valore di ritorno è `void`[5], e il suo scopo è quello di produrre un effetto nel terminale, dove verrà stampato un quadrato

5: `void` in inglese significa infatti *vuoto*, cioè non contiene nulla da ritornare.

di asterischi di dimensione pari al parametro `lato`. Se ad esempio venisse chiamata con un parametro di valore 5, il risultato a video sarebbe il seguente:

```
*****
*****
*****
*****
*****
```

Esistono anche funzioni che non hanno uno scopo esclusivamente didattico, come ad esempio quelle legate alla grafica, dove spesso si trova un funzione di nome `init` o similare, il cui scopo è appunto inizializzare la scheda grafica e comunicare al sistema operativo che il programma utilizzerà delle risorse per «disegnare» sullo schermo.

4.4. Passaggio di parametri per riferimento

Negli esempi precedenti è stato sfruttato il passaggio di valori *per valore*[6], che può essere pensato come il modo «naturale»[7] di passare i valori a una funzione in C++.

Sebbene questo meccanismo sia quello corretto in moltissimi casi, poiché *disaccoppia* le variabili che vengono passate da quelle che si trovano nella funzione, rendendola priva di «effetti collaterali» (*side effects* in inglese), ci sono dei casi in cui questo tipo di passaggio non è adatto.

Si supponga, ad esempio, di voler fare una funzione che scambi i valori tra due variabili: una tal funzione potrebbe essere molto utile negli algoritmi di ordinamento, nei quali le due operazioni principali sono il *confronto* e, appunto, lo *scambio*. Un'implementazione «ingenua» potrebbe essere la seguente:

6: Nel testo verranno usati come sinonimi i termini *passaggio per valore* e *passaggio per copia*.

7: Per naturale si intende che la sintassi di passaggio non coinvolge nessuna notazione o simbolo nuovo rispetto a una normale dichiarazione di variabile.

```
1  void scambio(int a, int b){
2      int temp = a;
3      a = b;
4      b = temp;
5  }
```

Listing 4.4: Implementazione dello scambio di due variabile con passaggio per copia.

Anche se, apparentemente, il codice è corretto, implementando il classico scambio tra due variabili utilizzando un'ulteriore variabile di appoggio, se si provasse ad utilizzarlo ci si accorgerebbe immediatamente che non funziona, come evidenziato dal codice

```
1  int main(){
2      int a = 5, b = 8;
3      std::cout << "Prima dello scambio: ";
```

Listing 4.5: Utilizzo della funzione scambio con passaggio per copia.

```
4         std::cout << "a = " << a << " b = " << b << std::
      endl;
5         scambio(a, b);
6         std::cout << "Dopo lo scambio: ";
7         std::cout << "a = " << a << " b = " << b << std::
      endl;
8         return 0;
9     }
```

Infatti l'esecuzione lascerebbe inalterate le variabili `a` e `b` presenti nel `main`, poichè la funzione scambierebbe i valori presenti nei suoi parametri `a` e `b`.

Per ottenere l'effetto voluto, il C++ introduce una nuova sintassi che permette il *passaggio per riferimento*, cioè quello che succede è che, stavolta, non vengono passate delle copie dei parametri attuali, ma la funzione agirà direttamente su quelle variabili.

Per questo scopo viene utilizzato il simbolo `&`, che è lo stesso presente nell'operatore AND logico `&&`, ma il cui significato è completamente diverso. Questo simbolo viene premesso ai parametri formali che si vuole vengano passati per riferimento e il compilatore effettuerà l'opportuno processo di traduzione che avrà come effetto di ottenere lo scambio delle variabili come inizialmente desiderato.

Listing 4.6: Implementazione dello scambio di due variabile con passaggio per riferimento.

```
1     void scambio(int &a, int &b){
2         int temp = a;
3         a = b;
4         b = temp;
5     }
```

Riprovando ad eseguire il codice 4.5 utilizzando la funzione definita nel listato 4.6, l'effetto voluto sarà esattamente quanto ci si proponeva.

Oltre a questo esempio ci sono ovviamente altre situazioni, che verranno presentate nei prossimi capitoli, nelle quali l'utilizzo del passaggio per riferimento è la soluzione giusta per il problema. Limitandosi a quanto visto finora si utilizza il passaggio per riferimento:

1. quando si vuole che la funzione modifichi i valori contenuti nei parametri attuali, come nell'esempio dello scambio
2. quando si vuole che la funzione ritorni più di un valore: siccome il meccanismo del valore di ritorno prevede che solo una cosa possa essere ritornata, si può mimare l'effetto di ritornare più valori «inserendo» questi valori nelle variabili passate per riferimento.

Per comprendere meglio questa seconda situazione si supponga di voler definire una funzione che calcoli le due soluzioni di un'equazione di secondo grado. Tale funzione non sarebbe ovviamente in grado di ritornare entrambe le soluzioni e quindi una possibilità per risolvere il problema è quello di progettare la funzione con la seguente firma:

```
1    bool risolvi(float a, float b, float c,
2                 float &x1, float &x2);
```

Listing 4.7: Firma della funzione per il calcolo delle soluzioni di un'equazione di secondo grado.

Come si può vedere, i primi tre parametri sono i coefficienti dell'equazione di secondo grado e vengono passati per copia, in quanto non si ha nessuna necessità di modificarli. Gli ultimi due, invece, vengono passati per riferimento: in questo modo la funzione, dopo che nel suo corpo avrà calcolato i due valori, li inserirà nei parametri `x1` e `x2` che quindi, all'uscita della funzione, conterranno le due soluzioni.

Si può vedere nel codice 4.8 l'esempio completo: alla riga 1 viene definita una funzione di supporto per il calcolo del *delta* della formula per il calcolo delle soluzione dell'equazione.

Successivamente, nella funzione `risolvi`, viene utilizzata questa funzione per calcolare il delta, ed eventualmente, se dovesse essere negativo (riga 8), viene ritornato il valore **`false`** per indicare al chiamante che non è possibile calcolare le soluzioni[8].

Le righe 10 e 11 applicano le note formule[9] per trovare le due soluzioni, che vengono «inserite» nelle variabili `x1` e `x2`: essendo queste variabili passate per riferimento, gli assegnamenti produrranno l'effetto di modificare anche i valori contenuti nei parametri attuali e, quindi, all'interno del `main` sarà, ad esempio, possibile stamparli o farne qualunque a cui si sia interessati.

8: Nel caso invece il delta fosse positivo le soluzioni verranno calcolate e la funzione ritornerà il valore **`true`** per indicare la buona riuscita del calcolo.

9: La funzione `sqrt` calcola la radice quadrata di un numero, appartiene alla libreria matematica del C++ e verrà ripresa in seguito.

Listing 4.8: Esempio completo per il calcolo delle soluzioni di un'equazione di secondo grado.

```
float delta(float a, float b, float c){
    return b*b - 4*a*c;
}

bool risolvi(float a, float b, float c,
        float &x1, float &x2){
    float d = delta(a, b, c);
    if (d < 0)
        return false;
    x1 = (- b + sqrt(d))/(2*a);
    x2 = (- b - sqrt(d))/(2*a);
    return true;
}

int main() {
    float a , b, c;
    float x1, x2;
    std::cout << "Risoluzione di un'equazione di
secondo grado" << std::endl;
    std::cout << "Inserire i tre coefficienti a, b, c:"
    std::cin >> a >> b >> c;
    if (risolvi(a, b, c, x1, x2)){
        std::cout << "x1 = " << x1 << std::endl;
        std::cout << "x2 = " << x2 << std::endl;
    }
    else{
        std::cout << "Non ci sono soluzioni nel campo
dei reali" << std::endl;
    }
    return 0;
}
```

Come progettare una funzione

Per progettare una funzione è conveniente seguire i seguenti passi:

1. decidere qual è lo scopo della funzione
2. scegliere un nome opportuno, valido da un punto di vista sintattico e che aiuti a comprenderne lo scopo
3. scegliere se ritorna qualcosa o no (`void`), e nel caso lo ritorni, scegliere il tipo corretto (`int`, `float`, `double`, ecc.)
4. individuare i parametri, cioè le informazioni che sono necessarie alla funzione per svolgere il proprio compito.
5. decidere il tipo e l'ordine dei parametri, sempre con lo scopo di rendere un eventuale utilizzo della funzione il più intuitivo possibile
6. documentare il funzionamento, per quelle parti che non possono essere comprese guardando semplicemente gli aspetti descritti in precedenza.

Dove possibile deve essere *indipendente* dal contesto in cui si trova: un semplice metodo per verificare che soddisfi questa condizione è quello di pensare a cosa succederebbe che venisse utilizzata in un programma diverso da quello in cui è stata definita. Se fosse possibile utilizzarla come una *blackbox*, limitandosi quindi a conoscerne la firma, allora la funzione può essere considerata indipendente. Se invece dipendesse da aspetti esterni come l'input/output oppure dall'ordine con cui viene chiamata rispetto ad altre funzioni, allora avrebbe delle dipendenze: questo non è di per sé qualcosa che deve essere evitato, ma deve essere documentato opportunamente.

4.5. Esempi di funzioni

In questo paragrafo verranno mostrate sia alcune funzioni di libreria, per vedere come sono state progettate e come vanno utilizzate, sia un esempio completo che mostrerà come progettare delle proprie funzioni per risolvere un problema complesso.

Funzioni della libreria del C++

La libreria matematica del C++ fornisce delle funzioni che coprono le esigenze più comuni dei programmi che hanno a che fare con i numeri. Ne verranno qui analizzate tre, che possono tornare utili e sono sufficientemente semplici da utilizzare.

Calcolo della radice quadrata

La funzione per il calcolo della radice quadrata, come già visto in precedenza, si chiama `sqrt` e la sua firma è la seguente:

```
double sqrt(double x);
```

Come si può vedere è un ottimo esempio di *blackbox*, in quanto richiede un valore `double` come parametro e restituisce un numero, sempre `double`, come risultato: chi la utilizza non ha nessuna necessità di sapere che algoritmo è stato implementato per poterla usare e produrre il risultato cercato. Un esempio di una chiamata è il seguente:

Listing 4.9: Esempio di utilizzo della funzione `sqrt`.

```
1 #include <iostream>
2 #include <cmath>
3
4 int main() {
5     double n, r;
6     std::cout << "Inserire il valore positivo di cui si
      vuole sapere la radice quadrata:";
7     std::cin >> n;
```

```
8       r = sqrt(n);
9       std::cout << "La radice quadrata di "  << n << " è " <<
         r << std::endl;
10      return 0;
11  }
```

dove, se l'utente dovesse inserire il valore 3, il programma indicherà che la radice quadrata di 3 è 1.73205.

Da evidenziare che in generale una funzione è pensata per lavorare sotto ipotesi ragionevoli: nel caso della funzione di libreria `sqrt` l'ipotesi fatta dai progettisti è che il valore passato sia maggiore o uguale a zero. Se questo non si dovesse verificare, la funzione produrrà un valore speciale indicato in output come `nan`, che sta per *Not A Number*. Se si vuole evitare questa situazione sarà il chiamante che dovrà fare gli opportuni controlli prima della chiamata, per fare in modo che questo non succeda.

Elevamento a potenza

La funzione per l'elevamento a potenza si chiama `pow` e la sua firma è la seguente:

```
double pow(double base, double exponent);
```

Anche in questo caso, l'algoritmo per il calcolo potrebbe essere complesso, anche solo per il fatto che, essendo l'esponente un numero con virgola, non potrà essere realizzato utilizzando delle moltiplicazioni ripetute. La complessità viene comunque nascosta all'interno della *blackbox*, quindi utilizzare questa funzione evita qualsiasi sforzo implementativo e, in una sola riga, si ottiene l'elevamento a potenza, fornendo una qualsiasi coppia di numeri, come si può vedere nell'esempio seguente:

Listing 4.10: Esempio di utilizzo della funzione `pow`.

```
1   #include <iostream>
2   #include <cmath>
3
4   int main() {
5       double b, e, r;
6       std::cout << "Inserire il valore della base:";
7       std::cin >> b;
8       std::cout << "Inserire il valore dell'esponente:";
9       std::cin >> e;
10      r = pow(b, e);
11      std::cout << b << " elevato a "  << e << " è " << r <<
        std::endl;
12      return 0;
13  }
```

dove, inserendo come base 2.3 e come esponente 4.5 si otterrà 42.44.

Valore assoluto

La funzione per il calcolo del valore assoluto[10] di un numero, si chiama `abs` e la sua firma è la seguente:

```
int abs(int n);
```

A differenza dei due esempi precedenti, in questo caso l'implementazione è banale, basta un semplice `if` per invertire il valore nel caso sia negativo. Nonostante ciò questa funzione appartiene all'insieme delle funzioni di libreria perché assolve comunque a una necessità spesso presente nei programmi che trattano numeri e, inoltre, aumenta la leggibilità del programma, rispetto all'utilizzo di un generico `if`:

10: Si ricorda che il *valore assoluto* di un numero è il suo valore «privato» del segno. In matematica questa funzione viene spesso indicata come *modulo*: attenzione che con il termine modulo in programmazione si intende il resto intero di una divisione.

Listing 4.11: Esempio di utilizzo della funzione `abs`.

```
#include <iostream>
#include <cmath>

int main() {
    int a, b;
    std::cout << "Inserire un numero positivo:";
    std::cin >> a;
    std::cout << "Inserire un numero negativo:";
    std::cin >> b;
    std::cout << "Il valore assoluto di "  << a << " è " <<
     abs(a) << std::endl;
    std::cout << "Il valore assoluto di "  << b << " è " <<
     abs(b) << std::endl;
    return 0;
}
```

Un'ultima osservazione per concludere: il tipo dei parametri è stato scelto dai progettisti della libreria, in modo da rendere le funzioni le più generali possibili. Nel caso venissero passati dei parametri di tipo differente da quelli richiesti o il valore di ritorno venisse assegnato a una variabile di tipo differente, valgono tutte le considerazioni fatte nella Sezione 6 (Conversione di tipo e casting) sulle conversioni implicite e su eventuali perdite di precisione.

Funzioni per la generazione di numeri pseudo-casuali

Un altro paio di funzioni che vengono usate spesso sono quelle che riguardano la generazione di numeri casuali, già anticipate nella Sezione 3.5 e che si trovano in appendice nella Sezione A.4.

La funzione che genera numeri *pseudo-casuali*[11] è la funzione `rand`, la cui firma è la seguente:

```
int rand();
```

La chiamata di questa funzione ritorna un numero intero positivo che non può essere minore[12] di 32767. Siccome la tipica necessità

11: Vengono definiti pseudo-casuali e non casuali perché l'algoritmo che li genera si limita a produrre una sequenza di numeri che ha alcune proprietà che mostrerebbe anche una vera sequenza casuale e che, per gli scopi didattici per cui verranno utilizzati, sono una buona approssimazione di una vera sequenza casuale. Per approfondimenti si veda `https://www.random.org/`.

12: Lo standard C++ garantisce che qualsiasi compilatore debba soddisfare questa condizione e molto spesso il valore massimo è proprio 32767.

di chi usa la funzione è quella di avere un preciso insieme di valori compresi tra un minimo e un massimo, si ricorre alla formula che si può vedere nel codice seguente:

Listing 4.12: Esempio di utilizzo della funzione `rand`.

```
#include <iostream>
#include <cmath>

int main() {
    int min, max;
    std::cout << "Inserire il valore minimo:";
    std::cin >> min;
    std::cout << "Inserire il valore massimo:";
    std::cin >> max;
    for (int i = 0; i < 10; ++i) {
        std::cout << rand() % (max - min + 1) + min << std
    ::endl;
    }
    return 0;
}
```

che sfrutta le proprietà dell'operatore *modulo* (%) per ottenere un numero compreso nell'intervallo $[min, max]$, dove, ad esempio, scegliendo come valore minimo il numero 1 e come valore massimo il 6, si otterrà la simulazione dei lanci di un dado a 6 facce.

Provando ad eseguire questo codice ci si accorgerà che la sequenza dei numeri prodotti sembra essere sufficientemente casuale, il problema è che si ripete esattamente uguale tra un'esecuzione e l'altra del programma. Questo non è un problema, ma una caratteristica dei generatori di numeri pseudo-casuali, che può anche tornare comoda quando si sta *debuggando*[13] il programma e si vogliono avere dei valori sempre uguali. Chiaramente questa caratteristica rende piuttosto inutile il processo di introdurre casualità in un programma «vero», in quanto, dopo la prima esecuzione, l'esito sarebbe sempre lo stesso: si pensi ad esempio a un gioco di carte, con il mazzo mescolato sempre nello stesso identico modo, o un gioco di dadi su cui bisogna puntare sull'uscita di certi valori, sapendo che dopo la prima esecuzione, tutte le successive ripeteranno la stessa sequenza di lanci. Per fare in modo che ogni esecuzione del programma produca sequenze casuali e anche diverse tra di loro, bisogna utilizzare la funzione `srand`[14], la cui firma è la seguente:

13: Il termine *debuggare* deriva dall'inglese *debug* e indica l'azione svolta dai programmatori per eliminare errori nascosti nei propri programmi.

```
void srand (unsigned int seed);
```

Il parametro passato è quello che inizializza il seme e quindi non può essere un valore costante, in quanto, ovviamente, avendo sempre lo stesso seme, la sequenza prodotta sarà sempre la stessa. Il modo più semplice è quello di «leggere» il tempo di sistema con la funzione di libreria `time` e utilizzare quello per inizializzare il seme, in quanto il tempo sarà sempre diverso. Mettendo insieme quanto detto, sarà necessario fare la seguente chiamata, una volta

14: Il nome è una contrazione di *seed rand*, in quanto inizializza il *seme*, cioè il valore numerico, che poi verrà utilizzato come elemento di partenza della sequenza e ne determina l'esatta serie di valori.

sola, all'inizio del programma, ad esempio alla riga 6 del listato 4.12.

```
srand(time(NULL));
```

Interessante notare che queste due funzioni, la `rand` e la `srand`, sono legate fra di loro, in quanto la chiamata di `srand` influenzerà i valori prodotti da `rand` e quindi non rispettano strettamente la condizione di indipendenza indicata nella Sezione 9 (Come progettare una funzione): nonostante questo, la dipendenza è ben documentata nella libreria e risulta quindi utile e perfettamente accettabile.

4.6. Un esempio completo: calcolare la differenza tra date

Si supponga di voler scrivere una funzione che calcoli la differenza in giorni tra due date qualsiasi, ad esempio tra il 19 giugno 2022 e il 26 agosto 2023, che in questo esempio risulta essere di 433 giorni.

Seguendo le indicazioni espresse nella Sezione 9 (Come progettare una funzione), il progetto della funzione sarà:

1. decidere qual è lo scopo della funzione: calcolare la differenza tra due date espressa in giorni. Già a questo punto è necessario decidere se la differenza conterrà o meno il giorno finale, per poterlo documentare a favore di chi utilizzerà la funzione. In questo esempio non si terrà conto del giorno finale, quindi la differenza tra due giorni consecutivi sarà di un giorno.
2. scegliere un nome opportuno, valido da un punto di vista sintattico e che aiuti a comprenderne lo scopo: `differenza_date` sembra una buona scelta, conciso ma sufficientemente autodocumentante, chi la utilizzerà dovrebbe essere in grado di capirne subito lo scopo
3. scegliere se ritorna qualcosa o no (**`void`**), e nel caso lo ritorni, scegliere il tipo corretto (**`int`**, **`float`**, **`double`**, ecc.): siccome la differenza sarà un numero di giorni, il valore da ritornare sarà sicuramente un intero
4. individuare i parametri, cioè le informazioni che sono necessarie alla funzione per svolgere il proprio compito: la scelta più ovvia sarebbe quella di passare due date, ma, per quanto visto finora, il linguaggio non permette di dichiarare variabili di un ipotetico tipo *Data*[15]. L'unica possibilità è quindi di passare i singoli componenti di una data. cioè il giorno, il mese e l'anno delle due date

15: Si vedrà in seguito che questa lacuna verrà colmata da un costrutto del linguaggio che permetterà di definire nuovi tipi di dati non presenti nel linguaggio.

5. decidere il tipo e l'ordine dei parametri, sempre con lo scopo di rendere un eventuale utilizzo della funzione il più intuitivo possibile: come detto al punto precedente i parametri saranno 6 e l'ordine più naturale dovrebbe essere quello nel quale i primi 3 parametri sono rispettivamente il giorno, il mese e l'anno della prima data, mentre i secondi 3 sono il giorno. il mese e l'anno della seconda data. Chiaramente per tutti i parametri il tipo sarà `int`
6. documentare il funzionamento, per quelle parti che non possono essere comprese guardando semplicemente gli aspetti descritti in precedenza: ci sono diversi aspetti che devono essere ben documentati e su cui prima devono essere fatte delle scelte, che sono:
   - cosa succede se una data non è corretta, come ad esempio il 30/02/2023 oppure il 44/13/2023? Per semplicità, in questo esempio si farà l'ipotesi che le date inserite saranno sempre corrette, demandando un eventuale controllo al chiamante di questa funzione
   - la prima data deve necessariamente essere precedente alla seconda? Una scelta ragionevole è quella di fornire un numero positivo se la prima data è precedente alla seconda, un numero negativo viceversa, quindi la differenza tra il 19/06/2022 e il 22/06/2022 vale 3, mentre la differenza tra il 22/06/2022 e il 19/06/2022 vale -3
   - entro quale range di date la risposta sarà corretta? In altri termini, sarà possibile fare la differenza tra il giorno corrente e, ad esempio, il 25 dicembre 800? Sempre per semplicità, la funzione darà risultati corretti solo per date successive al 1° gennaio dell'anno 1 d.C.

Rispetto a quanto scelto, la firma della funzione sarà la seguente:

```
int differenza_date(int g1, int m1, int a1, int g2, int m2,
int a2);
```

e quindi ora sarà necessario progettare il corpo della funzione.

Una prima idea potrebbe essere quella di utilizzare delle semplici differenze tra i singoli giorni, mesi e anni e poi sommare quanto ottenuto, con una formula simile alla seguente:

$$(g2 - g1) + (m2 - m1) \times 30 + (a2 - a1) \times 365$$

Facendo però delle prove è facile verificare che una formula del genere funziona solo in alcuni casi, poichè non tiene conto del fatto che i mesi hanno lunghezze diverse e inoltre che alcuni anni sono bisestili. Solo per fare un esempio, la differenza tra il

27/02/2022 e il 01/03/2022, che sarebbe di 2, calcolata con quella formula darebbe 4. Il modo canonico di affrontare questo problema è invece quello di calcolare il numero dei giorni che passano da una certa data di «partenza», che qui sarà il 01/01/01, e le due date di interesse e successivamente fare la differenza tra i numeri così ottenuti. Chiamando d_0 la data di partenza, d_1 e d_2 le due date di cui si vuole avere la differenza in giorni e `giorni_passati` una funzione che ritorna la differenza in giorni tra d_0 e una data passata come parametro, si avrà:

$$\text{differenza} = \text{giorni\_passati}(d_2) - \text{giorni\_passati}(d_1)$$

la cui rappresentazione grafica può essere vista in figura Figura 4.5.

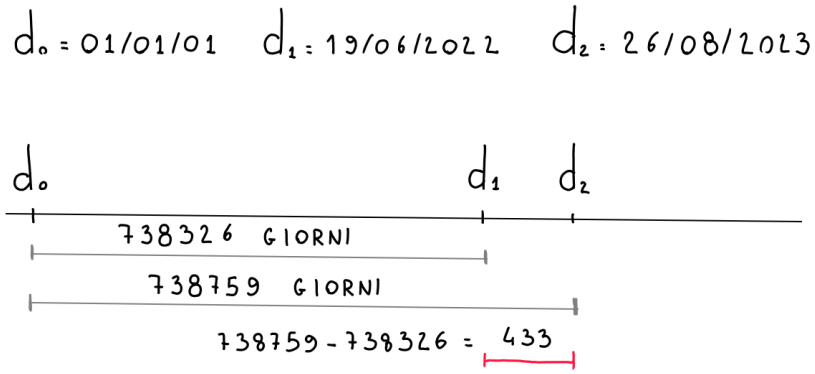


Figura 4.5.: Funzionamento del calcolo della differenza tra due date d_1 e d_2

In effetti, il problema sembra essersi complicato, perché di fatto dobbiamo fare due differenze anziché una, poiché anche `giorni_passati` al suo interno dovrà calcolare una differenza tra d_0 e la data passata come parametro. Il fatto però che d_0 sia una data fissata, permette di risolvere il problema una sola volta e poi applicarlo a d_1 e d_2.

Il problema si è quindi spostato alla costruzione di una funzione `giorni_passati` che faccia quanto richiesto in modo corretto e non sia troppo complessa da scrivere. Come già indicato, spesso la soluzione di un problema risiede nella soluzione di sottoproblemi più semplici, che, riflettendo, possono essere individuati nei seguenti:

1. verificare se un anno è bisestile o meno
2. calcolare i giorni che contiene un mese in un certo anno, poiché febbraio ha un numero di giorni variabile a seconda dell'anno in cui si trova
3. calcolare quanti giorni sono passati a partire all'1 gennaio fino ad arrivare al mese della data di interesse
4. calcolare quanti giorni sono passati a partire dall'anno 1 fino ad arrivare all'anno della data di interesse.

Questi due ultimi sottoproblemi permettono di calcolare il numero totale dei giorni come somma del contributi dati dagli anni, dai mesi e dai giorni che compongono le date di interesse. Esemplificando con la data 19/06/2022, il numero di giorni che la separano dall'01/01/01 sarà di 19 giorni per i giorni contenuti nella data, più 151 che sono i giorni dal 1 gennaio a tutto maggio compreso, più 738155 che sono i giorni dall'anno 1 fino a tutto il 2021 compreso.

Detto questo, la soluzione potrà essere sviluppata in piccole funzioni che poi verranno successivamente utilizzate per fornire la soluzione completa illustrata nel listato 4.17.

Listing 4.13: Funzione per verificare se un anno è bisestile o meno.

```
int bisestile(int anno)
{
    if (anno%400 == 0)
        return 1;
    if (anno%100 == 0)
        return 0;
    if (anno%4 == 0)
        return 1;
    return 0;
}
```

La funzione nel listato 4.13, le cui regole possono essere trovate all'Esercizio 10, ritorna 1 se l'anno è bisestile o 0 altrimenti. La scelta di questi due valori è vantaggiosa perché permette di utilizzare questa funzione nei listati 4.14 e 4.16 semplicemente sommando il valore di ritorno per poter aggiungere l'eventuale giorno in più negli anni bisestili.

Listing 4.14: Funzione che ritorna il numero di giorni contenuti in uno specifico mese.

```
int giorni_nel_mese(int mese, int anno)
{
    if (mese == 2)
        return 28 + bisestile(anno);
    if (mese == 11 || mese == 4 || mese == 6 || mese == 9)
        return 30;
    return 31;
}
```

La funzione `giorni_nel_mese` ritorna il numero di giorni contenuti nel mese, passato come parametro, dove 1 è gennaio, 2 è febbraio, ecc. Da notare che deve essere passato anche l'anno, per poter distinguere quando febbraio ha 28 piuttosto che 29 giorni.

Listing 4.15: Funzione che calcola il contributo dato dai mesi.

```
int giorni_mese(int mese, int anno)
{
    int giorni = 0;
    for (int i = 1; i < mese; i++) {
        giorni += giorni_nel_mese(i, anno);
    }
    return giorni;
}
```

La funzione `giorni_mese` implementa il punto 3 indicato in precedenza e la funzione `giorni_anno` implementa il punto 4.

Listing 4.16: Funzione che calcola il contributo dato dagli anni.

```
int giorni_anno(int anno)
{
    int giorni = 0;
    for (int i = 1; i < anno; i++) {
        giorni += 365 + bisestile(i);
    }
    return giorni;
}
```

Infine il listato 4.17 utilizza le funzioni viste in precedenza per risolvere il problema iniziale.

Listing 4.17: Funzioni per il calcolo della differenza tra due date.

```
int giorni_passati(int g, int m, int a){
    return giorni_anno(a) + giorni_mese(m,a) + g;
}

int differenza_date(int g1, int m1, int a1,
int g2, int m2, int a2)
{
    int giorni1, giorni2;
    giorni1 = giorni_passati(g1, m1, a1);
    giorni2 = giorni_passati(g2, m2, a2);
    return giorni2 - giorni1;
}
```

4.7. Esercizi

Studi

Esercizi elementari per prendere confidenza con le varie istruzioni presentate nel capitolo. Di tutte le funzioni richieste scrivere un breve programma che ne dimostri la correttezza.

1. Scrivere una funzione che faccia la somma tra due numeri interi e ne ritorni il risultato
2. Scrivere una funzione che faccia il prodotto tra un numero intero e un numero `float` e ne ritorni il risultato come `float`
3. Scrivere una funzione che confronti due parametri interi e restituisca `true` se il primo è maggiore del secondo, `false` altrimenti
4. Scrivere una funzione che controlli se tre parametri interi sono già ordinati dal più piccolo al più grande e nel caso ritorni `true`, altrimenti ritorni `false`
5. Scrivere una funzione che verifichi se il primo parametro intero è esattamente divisibile dal secondo e restituisca `true` nel primo caso, `false` nel secondo
6. Scrivere una funzione che restituisca la somma di tutti i numeri compresi tra due numeri interi **n** e **m** passati come parametri
7. Scrivere una funzione che restituisca il prodotto di tutti i numeri compresi tra due numeri interi **n** e **m** passati come parametri
8. Scrivere una funzione che restituisca la somma di tutti i numeri pari compresi tra due numeri interi **n** e **m** passati come parametri
9. Scrivere una funzione che restituisca quanti sono i numeri pari compresi tra due numeri interi **n** e **m** passati come parametri
10. Scrivere una funzione che restituisca la somma dei quadrati di tutti i numeri dispari compresi tra due numeri interi **n** e **m** passati come parametri
11. Scrivere una funzione che accetta un intero e restituisce `true` se il numero è pari e `false` altrimenti.
12. Scrivere una funzione che calcoli la potenza intera di un numero. La funzione avrà come parametri la base e l'esponente.
13. Scrivere una funzione che calcoli il fattoriale di un numero intero. Si ricorda che il fattoriale viene indicato con $n!$ ed è uguale al prodotto di tutti i numeri naturali minori o uguali a n, cioè $n! = i \times 2 \times 3 \times \cdots \times (n-1) \times n$.
14. Scrivere una funzione che accetta tre numeri interi come parametri e restituisce il numero massimo tra di essi.
15. Scrivere una funzione che calcola la media di tre valori in virgola mobile e restituisce il risultato.
16. Scrivere una funzione che calcola l'area di un cerchio dato il suo raggio come parametro.
17. Scrivere una funzione che stampi a video la tabellina del numero **n**, con **n** parametro intero.
18. Scrivere una funzione che accetta tre parametri interi: **inizio**, **n** e **passo**. La funzione deve stampare una sequenza di **n** numeri, che parta da **inizio** con il **passo** specificato. Se ad esempio i tre parametri fossero, nell'ordine 7, 4, 3, la funzione stamperà la sequenza 7, 10, 13, 16.
19. Scrivere una funzione che chieda all'utente di inserire un numero compreso tra 0 e 6 e se non fosse compreso nell'intervallo richiesto lo faccia reinserire fino a quando non viene soddisfatta la condizione. La funzione deve ritornare il numero inserito.
20. Scrivere una funzione che faccia lo scambio tra i due suoi parametri passati per riferimento

21. Scrivere una funzione che faccia inserire **N** numeri e calcoli il maggiore e il minore. La funzione ha come parametri **N** passato per valore e **maggiore** e **minore** passati per riferimento.
22. Scrivere una funzione che prende due numeri interi come argomenti e calcoli la loro somma e il loro prodotto. La funzione ha come parametri i due numeri passati per valore e **somma** e **prodotto** passati per riferimento.
23. Scrivere una funzione che calcoli le soluzioni di una equazione di secondo grado, che abbia come parametri i coefficienti **a, b, c** dell'equazione e le cui soluzioni vengano memorizzate nei due parametri **x1, x2** passati per riferimento
24. Scrivere una funzione che abbia come parametri le coordinate **x, y** di due punti nel piano e che calcoli le coordinate del loro punto medio **xm, ym**, passate per riferimento.

Esercizi

1. Si scriva una funzione che conti quanti sono i numeri primi compresi in un certo intervallo [a, b), con **a** e **b** numeri interi passati come parametri alla funzione (si supponga che a < b). Si suggerisce di definire prima la funzione `primo`, che verifica se un numero **N** passato come parametro è primo, per poterla utilizzare all'interno della funzione principale, in modo da semplificarne la scrittura. Si supponga inoltre che il parametro passato alla funzione `primo` sia sempre maggiore o uguale a 2, poichè negli altri casi non avrebbe senso porsi la domanda sulla primalità.
   Si scriva infine un semplice programma per verificare la correttezza della funzione scritta.
2. Scrivere una funzione che ritorni la *k*-esima cifra di un numero intero **n**, dove sia **k** che **n** devono essere passati come parametri. Se k fosse ≤ 0 oppure maggiore del numero di cifre di n, la funzione restituirà -1. Se ad esempio **n** valesse 4321 e **k** valesse 2, la funzione dovrà restituire 3.
   Si scriva infine un semplice programma per verificare la correttezza della funzione scritta.
3. Si scriva una funzione che, dati due numeri interi come parametri, ne calcoli il *minimo comune multiplo* (mcm), cioè il più piccolo multiplo comune ad entrambi.
   Si scriva successivamente un semplice programma per verificare la correttezza della funzione scritta.
4. Si scriva una funzione che, dati due numeri interi positivi come parametri, restituisca il *Massimo Comun Divisore* (MCD) tra i due, cioè il più grande numero che divide entrambi.
   Si utilizzi dapprima un algoritmo a **forza bruta**, che provi a fare divisioni fino a quando non trova la risposta corretta.
   Successivamente si riscriva la funzione utilizzando l'algoritmo di Euclide, dove, dati i due numeri **a** e **b**, si procede nel seguente modo:

   1 si calcola il resto della divisione tra **a** e **b**
   2 si verifica il valore del resto
   - se il resto è zero allora il valore di **b** è l'MCD cercato e l'algoritmo termina
   - se il resto è diverso da zero, alla variabile **a** viene assegnato il valore di **b** e alla variabile **b** viene assegnato il resto appena trovato. Si ricalcola il resto della divisione tra **a** e **b** e si ripete dal punto 2.

   Si scriva successivamente un semplice programma per verificare la correttezza della funzione scritta.
5. Scrivere una funzione che, dati come parametri 3 numeri che rappresentano i lati di un triangolo, **stampi a video** di che tipo di rettangolo si tratta, se *equilatero*, *isoscele* o *scaleno*.

Successivamente scrivere un programma che chieda all'utente di inserire i tre numeri e stampi il tipo di triangolo. Il programma deve proseguire fino a quando desidera l'utente.

6. Scrivere una funzione che, dato come parametro un numero intero positivo N, ritorni la differenza tra il quadrato della somma dei primi N numeri naturali e la somma dei quadrati dei primi N numeri naturali. Se ad esempio N fosse 3, la funzione dovrà ritornare il valore $(1+2+3)^2 - (1^2+2^2+3^2) = 36-14 = 22$. Scrivere un programma che, utilizzando la funzione scritta in precedenza, chieda all'utente di inserire un numero naturale N e calcoli la differenza sopra descritta, mostrandola all'utente. Il programma deve proseguire fino a quando desidera l'utente.
7. Scrivere una funzione che abbia come parametro un numero intero e restituisca `true` se il numero è un *palindromo* e `false` altrimenti. Un numero è palindromo se letto da destra a sinistra o da sinistra a destro è lo stesso numero: ad esempio, 12321 è palindromo, 12345 non lo è.
   Si scriva successivamente un semplice programma per verificare la correttezza della funzione scritta.
8. Scrivere una funziona che abbia come parametro un numero intero positivo **n** e restituisca il numero con le cifre invertite. Se ad esempio il numero fosse 1234, dovrà restituire 4321, se fosse 1200, restituirà 21 gli zeri iniziali non contano).
   Successivamente utilizzare la funzione creata per risolvere il problema indicato nell'esercizio precedente. verificando che si semplifica.
9. Scrivere una funzione che accetti un intero **n** e stampi un triangolo di asterischi con **n** righe, come nell'esempio mostrato qua sotto.

```
n = 5

    *
   ***
  *****
 *******
*********
```

   Si scriva successivamente un semplice programma per verificare la correttezza della funzione scritta.
10. Scrivere una funzione che accetti come parametro un numero intero **a** che rappresenta un anno e che restituisca `true` se l'anno è bisestile, `altrimenti`. Un anno è bisestile:

    - se è secolare (cioè multiplo di 100), allora deve essere divisibile per 400
    - se è non secolare, deve essere divisibile per 4.

    Ad esempio, il 1996, il 2000 e il 2004 erano anni bisestili, il 1900, il 1998 e il 2003 non lo erano.
    Si scriva successivamente un semplice programma per verificare la correttezza della funzione scritta.
11. Scrivere una funzione che, passate due date come parametri nella forma di giorno, mese, anno, restituisca la differenza in termini di giorni tra la prima e la seconda (il valore sarà positivo se la prima data è posteriore alla seconda, negativo viceversa).
    Si suggerisce di definire altre funzioni per recuperare il numero dei giorni contenuti in un mese e per sapere se un anno è bisestile (sfruttare quella definita nell'esercizio precedente).
    Si scriva successivamente un semplice programma per verificare la correttezza della funzione scritta.

12. Scrivere delle funzioni per convertire:
    - un numero decimale in un numero binario
    - un numero binario in un numero decimale
    - un numero decimale in un numero ottale
    - un numero ottale* in un numero binario

    Viene richiesto di usare delle variabili di tipo `long long`, sia come variabili che come valore di ritorno, per rappresentare ogni tipo di numero, sia decimale, che ottale, che binario: sarà l'insieme di cifre e l'utilizzo che se ne fa a stabilire che base ha. Se, ad esempio, l'utente deve inserire il numero binario `1011010`, dal punto di vista della variabile che lo contiene sarà comunque un numero intero in base dieci: dovrà essere la funzione a leggere ogni cifra come un bit in una certa posizione e, dovendo convertirlo in decimale, lo trasformerà nel numero `80`.
    Scrivere poi un programma che, utilizzando le funzioni scritte in precedenza e attraverso un menù, permetta all'utente di scegliere una conversione, gli faccia inserire un numero e lo mostri seguendo la conversione scelta.
13. Scrivere una funzione che permetta di fare la somma tra la durata di due impegni, espressi nella forma di ore e minuti. Se, ad esempio, le durate da sommare fossero di 3 ore e 15 minuti e 1 ora e 50 minuti, la funzione dovrebbe dare come somma 5 ore e 5 minuti.
    Si scriva successivamente un semplice programma che permetta all'utente di inserire un numero N di durate di impegni, con N a piacere, e che stampi la somma di tutte le durate inserite, utilizzando la funzione definita in precedenza.
14. Le funzioni trigonometriche come *seno* e *coseno* possono essere definite in modi diversi che permettono di calcolarne il valore in corrispondenza di un determinato angolo. Uno dei modi possibili è quello di utilizzare una *serie di Taylor*, che viene matematicamente definita nel seguente modo:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

    che, tradotto in italiano, indica che il seno di un certo valore x può essere ottenuto sommando tra loro una serie infinita di termini, della forma indicata nella formula, dove il valore di n va sostituito in ogni termine con un numero intero che parte da 0 e arriva, appunto, a infinito. Siccome non è possibile sommare infiniti termini, basta in generale fermarsi a un certo numero N, scelto dall'utente, per ottenere un'approssimazione del valore reale: più il valore di N sarà grande, migliore sarà l'approssimazione†.
    Si scriva quindi una funzione che, avendo come parametro un angolo x, espresso in *radianti*, e un numero intero N, calcoli il valore del seno di x usando la formula mostrata in precedenza, fermandosi alla somma dei primi N termini.
    Si scriva successivamente un semplice programma che, fatto inserire un angolo dall'utente espresso in gradi, lo trasformi in radianti e calcoli il valore del seno usando la funzione appena definita, variando il valore di N a partire da 1, fino ad arrivare a 10. Si verifichi quindi che il risultato ottenuto dalla funzione diventa migliore al crescere di N, confrontandolo con il valore prodotto dalla funzione di libreria `sin`.
    Si ricorda che il rapporto tra gradi e radianti è espresso dalla formula:

$$\pi \text{ radianti} = 180 \text{ gradi}$$

* Un numero ottale è un numero in base 8, dove, quindi, le sue cifre sono dallo 0 al 7

† Siccome nell'espressione compare il fattoriale, come già visto in altri esercizi, il valore di n non potrà essere troppo grande per evitare problemi di *overflow*.

15. Si scriva una funzione che, dati come parametri il numero di ore lavorative ordinarie e il numero di ore straordinarie lavorate in un mese, permetta di calcolare lo stipendio lordo guadagnato e le tasse che verranno detratte.
    Le regole per il calcolo dello stipendio e delle tasse da pagare sono le seguenti:
    Stipendio

    - ogni ora ordinaria è pagata 15.4 euro
    - ogni ora straordinaria è pagata 22.8 euro

    Tasse

    a) lo stipendio fino a 5000 euro non è tassato
    b) per la successiva parte di stipendo fino a 22000 euro, le tasse sono pari al 22%
    c) per la successiva parte di stipendio fino a 50000 euro, le tasse sono pari al 33%
    d) per la parte di stipendio eccedente i 50000 euro, le tasse sono pari al 38%

    Scrivere successivamente un programma che permetta all'utente di inserire le ore ordinarie e straordinarie lavorate da un dipendente in un mese e stampi il suo stipendio lordo e le tasse da detrarre. Il programma deve permettere la ripetizione del calcolo per un numero di dipendenti a piacere, fermandosi quando l'utente vorrà finire con i calcoli.
    Modificare infine la funzione e il programma in modo che il valore delle ore ordinarie e straordinarie possa essere impostato dall'utente all'inizio del programma ed, eventualmente, possa anche essere modificato in modo che sia diverso per dipendenti diversi. Si cerchi di fare questa modifica in modo che il codice rimanga il più possibile simile a quella della prima versione.

Progetti

Calcolatrice tra frazioni

Si scrivano le seguenti funzioni che, date due frazioni nella forma di coppie numeratore/denominatore, calcolino il risultato dell'operazione specificata. Le funzioni dovranno inserire il risultato in ulteriori due parametri passati per riferimento.

- **somma**: fa la somma tra due frazioni
- **differenza**: fa la differenza tra due frazioni
- **prodotto**: fa il prodotto tra due frazioni
- **divisione**: fa la divisione tra due frazioni

Scrivere poi due funzioni di supporto con il seguente scopo:

- **semplifica**: ricevuti come parametri numeratore e denominatore di una funzione, riduce la frazione ai minimi termini. Se ad esempio il numeratore valesse 15 e il denominatore 21, dopo la chiamata di questa funzione diventerebbero rispettivamente 5 e 7.
- **stampa**: stampa la funzione nella forma `numeratore/denominatore`. Se ad esempio il numeratore valesse 3 e il denominatore 7, la funzione stamperebbe `3/7`. Attenzione ai casi particolari, che vanno stampati nel modo solito della matematica, quindi:
  - numeratore = 3, denominatore = 1 deve essere stampato come 3, **non** come 3/1
  - numeratore = 3, denominatore = -4 deve essere stampato come `-3/4`, **non** come 3/-4
  - numeratore = -3, denominatore = -4 deve essere stampato come 3/4, **non** come -3/-4

- numeratore = 0, denominatore = 4 deve essere stampato come 0, **non** come 0/4

Si può supporre che il denominatore sia sempre diverso da zero, oppure fare dei controlli che impediscano l'inserimento dello zero come valore per il denominatore.

Si scriva successivamente un programma che, tramite un menù, permetta di fare le quattro operazioni tra frazioni, sfruttando le funzioni descritte in precedenza, con i valori delle frazioni inseriti dall'utente. Il programma deve permettere di fare operazioni fino a quando l'utente desidera. Viene lasciata libertà nei dettagli, la cosa importante è che si cerchi di organizzarlo nel modo più ordinato possibile.

Gestione di un parcheggio

Un garage fa pagare una tariffa minima di 2,00 euro per parcheggiare fino a tre ore, più 0.50 all'ora per ogni ora o parte di essa oltre le tre ore. Il costo massimo per una giornata è comunque di 10,00 euro. Si supponga che nessuna macchina resti parcheggiata per più di 24 ore.

Scrivere un programma che calcoli e stampi i costi del parcheggio per ciascuno degli N clienti che hanno parcheggiato le loro auto in questo garage.

Versione semplice

Far inserire le ore di parcheggio per ogni cliente. Il programma dovrà stampare i risultati in un formato tabellare e dovrà calcolare e stampare il totale degli incassi di ieri. Il programma userà la funzione `calcolaCosti` per determinare il costo per ogni cliente. Decidere in maniera autonoma quali saranno i parametri e l'eventuale valore di ritorno della funzione.

L'output finale deve avere un formato simile a questo:

Macchina	Ore	Costo
1	1.5	2.00
2	4.5	3.00
3	24.0	10.00
TOTALE	29.5	14.50

Versione un po' più complessa

Al posto di inserire le ore del parcheggio, va inserito l'orario di ingresso e quello di uscita di ogni macchina, per il resto il programma è lo stesso. Utilizzare una funzione di supporto per la trasformazione orario ingresso - orario uscita nel numero di ore parcheggiate, in modo poi da utilizzare la funzione `calcolaCosti` definita in precedenza per il calcolo dei costi. Decidere in maniera autonoma quali saranno i parametri e l'eventuale valore di ritorno della funzione.

5. I vettori

Introduzione

Spesso in Informatica è necessario memorizzare un insieme di dati omogenei, ognuno dei quali rappresenta un valore specifico: si pensi ad esempio ai voti di un compito in classe, ai tempi fatti ad ogni giro di pista da una macchina da corsa, alle temperature registrate nel corso di un anno, ecc.

Tutte queste situazioni sono caratterizzate dalla necessità di memorizzare un numero N di valori, dove in generale N non è noto a priori e può essere anche molto grande: la possibilità di memorizzarli permetterà di recuperarli in seguito ed eseguire delle operazioni su di essi.

Per gestire queste situazioni i linguaggi di programmazione mettono a disposizione degli oggetti chiamati *vettori* [1], il cui scopo è appunto quello di memorizzare un numero arbitrario di valori della stesso tipo.

1: In questo testo verrà preferibilmente usato il termine *vettore*: in altri contesti o linguaggi viene utilizzato il termine *array* per indicare lo stesso concetto, quindi potranno anche essere usati come sinonimi.

In un certo senso, dove le strutture di controllo iterative, come il `while` e il `for`, permettono di ripetere una serie di operazioni un numero arbitrario di volte, i vettori permettono di memorizzare un numero arbitrario di variabili, aprendo quindi nuove possibilità al programmatore.

Nel linguaggio C++ esistono due tipi di vettori:

- i vettori *nativi del linguaggio*, ereditati dal C e mantenuti per retrocompatibilità[2], che sono il modo basilare per implementare il concetto di vettore,
- la più moderna classe `vector`, che risolve alcune problematiche dei vettori nativi, rendendoli robusti e di più semplice utilizzo: questa non è tecnicamente una parte del linguaggio, bensì una *libreria standard* compresa in tutte le implementazioni moderne del C++.

2: Per retrocompatibilità, in questo contesto, si intende la caratteristica di un sorgente vecchio di rimanere valido anche se utilizzato in una nuova versione del linguaggio.

5.1. I vettori

Una possibile formalizzazione del concetto di vettore appena presentato è la seguente:

Definizione di vettore

Un **vettore** è una *sequenza* di elementi *omogei*, cioè dello stesso tipo, individuati da un *indice* numerico intero. Per sequenza si intende che ogni elemento ha un elemento che lo precede e uno che lo segue, tranne evidentemente il primo e l'ultimo. Questo permette di individuare un singolo elemento utilizzando la sua *posizione*, che può essere indicata utilizzando un intero positivo, definito appunto *indice*.

Per capire come mai è necessario introdurre questo nuovo costrutto del linguaggio, si pensi ai due esempi seguenti:

Primo esempio: dati letti da un sensore

Problema: dati N numeri che rappresentano i valori di PM10[3] giornalieri letti da un sensore in un certo periodo, stabilire quanti valori sono superiori alla media del periodo di osservazione.

Motivo: prima di poter contare quanti valori sono superiori alla media del periodo, è appunto necessario calcolare la media, facendosi dare in input la sequenza di valori. Se poi non venissero memorizzati, cosa che per il calcolo della media non sarebbe necessaria, bisognerebbe richiederne le reimmissione, creando sicuramente fastidio all'utente del programma.

3: Un tipo di polveri sottili presenti nell'aria e misurate in $\mu g/m^3$.

Secondo esempio: ordinamento delle misure in una gara di salto in lungo

Problema: In una gara di salto in lungo vengono memorizzate le lunghezze dei salti dei vari atleti e successivamente bisogna ordinarle dalla misura migliore a quella peggiore.

Motivo: anche in questo caso, per poter procedere con l'ordinamento, è prima necessario acquisire tutti i valori, poiché chiaramente non potrebbero essere confrontati l'uno con l'altro per stabilire quale va prima e quale dopo senza averli tutti disponibili. Una volta acquisiti, ogni valore dovrebbe essere confrontato con ogni altro[4] e quindi dovrebbe essere possibile accedere ad ognuno di essi, indipendentemente dal fatto che possano essere poche decine, come nell'esempio corrente, o parecchi milioni.

4: Esistono diversi algoritmi di ordinamento, distinti per i metodi utilizzati e per l'efficienza con cui operano, tutti sono comunque accomunati dal fatto che bisogna effettuare dei confronti tra gli elementi e poi, eventualmente, procedere con degli scambi di posizione, dove necessario.

5.2. I vettori nativi del linguaggio

Data la definizione Definizione 5.1, le caratteristiche principali dei vettori nativi, che verranno approfondite in seguito, sono:

- ogni elemento è individuato da un indice, un intero che parte da 0 e **può** arrivare fino alla dimensione del vettore meno uno: se ad esempio un vettore fosse formato da 100 elementi, gli indici andrebbe da 0 a 99 incluso
- occupano celle di memoria consecutive, il che li rende estremamente efficienti nel restituire il valore di un elemento nella posizione *n*-esima, indipendentemente dal valore di *n* e dalla dimensione del vettore[5] e quindi si dice che l'operazione di lettura e scrittura di un elemento in un vettore ha sempre lo stesso *costo*.
- la dimensione di un vettore *nativo* è costante e in C++ viene stabilita all'atto della sua dichiarazione.

5: Spesso viene indicato questo tipo di accesso come *accesso casuale*, contrapposto all'*accesso sequenziale*, in cui invece il costo di accesso a un elemento dipende dalla posizione in cui si trova.

Mentre le prime due caratteristiche sono comuni anche ai `vector`, in quanto inerenti all'idea astratta di vettore, l'ultima evidenzia la natura essenzialmente *statica* dei vettori nativi, intesi come contenitori la cui grandezza non può variare in *run-time*, cioè mentre il programma è in esecuzione.

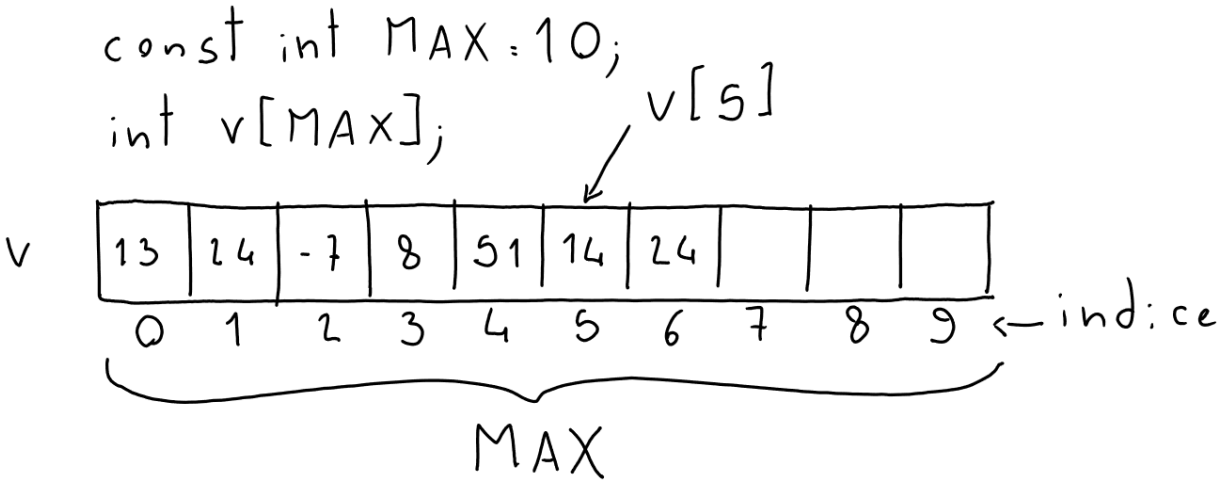


Figura 5.1.: Rappresentazione di un vettore

In figura Figura 5.1 si può vedere un vettore di dimensione 10, denominato v, contenente una serie di 7 numeri interi[6]. La riga di numeri sotto ad ogni cella rappresenta gli *indici* che individuano le posizioni dei valori in esso contenuti, dove ad esempio la cella di indice 5 conterrà il valore 14.

6: Le ultime tre caselle sono vuote per indicare che quelle tre caselle non sono utilizzate dal programma: nella realtà però le celle di un vettore conterranno sempre dei valori, eventualmente casuali.

Dichiarazione di un vettore

Come per le variabili «normali», anche per i vettori il primo passaggio per poterli utilizzare è quello di dichiararli. La dichiarazione non è molto diversa da quella delle variabili viste finora e si presenta in questo modo:

```
1  int main() {
2      const int MAX = 10;
3      int v1[MAX]];
4      float v2[MAX];
5      //resto del programma
6      return 0;
7  }
```

Listing 5.1: Dichiarazione di un vettore di interi e di un vettore di float, ambedue composti da 10 caselle.

Come si può vedere nel listato 5.1, la struttura generale di dichiarazione è la seguente:

Struttura della dichiarazione di un vettore

```
tipo nome[dimensione];
```

Le tre parti che la compongono sono quindi:

1. il *tipo*, che, come già visto per le variabili, può assumere i valori `int`, `char`, `float`, `double`, `bool`.
2. il *nome* del vettore, che, essendo comunque una variabile, deve rispondere alle considerazioni di correttezza sintattica già viste in precedenza e deve essere significativo rispetto al contesto di dichiarazione
3. una coppia di parentesi quadre che deve contenere al suo interno un numero intero positivo che stabilisce la *dimensione* del vettore.

Rispetto a quest'ultimo punto vanno fatte un paio di considerazioni:

1. una volta stabilita la dimensione al momento della dichiarazione, questa rimane fissa per tutta la durata del programma.
2. è una buona prassi indicare la dimensione utilizzando una variabile di tipo *costante* con la parola chiave `const`, in modo che sia più significativa rispetto a un «magic number[7]». In questo modo se eventualmente si volesse dare un valore diverso, sarà chiaro dove modificare questo valore, che è il punto in cui viene dichiarata questa variabile.

Rispetto al punto 1 va posta molta attenzione al fatto che la dimensione indicata al momento della dichiarazione non necessariamente coincide con il numero di elementi che verranno inseriti nel vettore durante l'esecuzione del programma, che al massimo saranno

7: In programmazione si indica come *magic number* un numero puro che si trova all'interno di un codice e il cui significato risulterebbe oscuro per chi dovesse successivamente fare modifiche o comunque lavorare sul programma.

coincidenti con la dimensione, ma potrebbero anche essere di meno, mai di più.

Dichiarazione e inizializzazione immediata

Esistono dei casi, soprattutto negli esempi didattici, in cui può essere comodo assegnare un insieme di valori ad un vettore direttamente all'atto della dichiarazione. Il C++ prevede questa possibilità utilizzando la sintassi mostrata in questo esempio:

```
int v[] = {2, 5, 6, 9};
```

Come si può vedere, nella dichiarazione del vettore stavolta manca l'indicazione della dimensione massima: questo perché sarà il compilatore a fare in modo che lo spazio in memoria riservato al vettore sia esattamente quello necessario per ospitare gli elementi con cui viene inizializzato, che in questo esempio sono 4. Questa notazione compatta quindi creerà un vettore di nome `v`, di dimensione 4, le cui caselle saranno inizializzate con i valori presenti nella lista di inizializzazione.

Lettura e scrittura degli elementi di un vettore

Come per le variabili «singole» viste finora, una delle azioni che si vogliono spesso compiere dei programmi, è quella di inserire all'interno di un vettore dei valori acquisiti da tastiera e di mostrare a video i valori in esso contenuti.

Nel caso dei vettori, siccome ogni singolo elemento di un vettore è una variabile, può essere trattato come già visto. Solitamente la differenza risiede nel fatto che, essendo un vettore un insieme di variabili tra loro correlate, le operazioni verranno spesso fatte su tutti gli elementi, *iterando* le letture o le scritture su ogni elemento, come si può vedere nel listato 5.2.

Listing 5.2: Esempi di input/output su un vettore.

```
1  int main() {
2      const int MAX = 4;
3      int v[MAX];
4      std::cout << "Inserisci " << MAX << " numeri interi."
       << std::endl;
5      for (int i = 0; i < MAX; ++i) {
6          std::cout << "Numero " << i + 1 << " : ";
7          std::cin >> v[i];
8      }
9      //Eventuali operazioni sul vettore
10     std::cout << "Mostra a video i numeri inseriti" << std
       ::endl;
11     for (int i = 0; i < MAX; ++i) {
```

```
12          std::cout << "Numero " << i + 1 << " : " << v[i] <<
        std::endl;
13      }
14      return 0;
15  }
```

Ogni elemento viene individuato da un indice, che, come già detto, parte da 0 per indicare la prima posizione, e quindi l'indice viene incrementato fino al valore dell'ultima posizione, in questo caso `MAX - 1`, scorrendo tutte le 4 caselle presenti nell'esempio. A posteriori si può anche capire perché si preferiva, anche in precedenza, far iniziare il valore del contatore del ciclo `for` da 0 anziché da 1: in questo modo la scrittura del ciclo si adatta naturalmente all'iterazione di tutti gli elementi di un vettore.

Un esempio: lo scarto quadratico medio

In statistica lo *scarto quadratico medio*, indicato con la lettera greca σ, viene calcolato su un insieme di valori e indica quanto questo insieme è più o meno «compatto» intorno alla media. Questa informazione permette di distinguere tra insiemi in cui la media è uguale, ma la distribuzione dei singoli valori potrebbe essere molto differente, dove valori più alti indicano una maggior dispersione, mentre valori più bassi indicano una maggiore compattezza intorno alla media. Per fare un esempio: avendo due studenti, il primo che ha ricevuto i voti 6, 8, 4 e il secondo 6, 6, 6, la media è uguale per entrambi e pari a 6, ma il primo studente ha chiaramente dei voti maggiormente dispersi, mentre il secondo, che è un caso limite, presenta viceversa una compattezza elevata.

Per definire in termini quantitativi questo parametro, la formula da utilizzare è la seguente:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

dove x_i è ognuno dei valori usati per calcolarla e $\overline{x}$ è la media aritmetica di quei valori. Espressa come un algoritmo, la formula può essere «raccontata» in questo modo:

- per ogni valore, si prenda il quadrato della differenza con la media e lo si accumuli per ottenere la somma di tutti i quadrati
- si divida la somma ottenuta per N
- si faccia la radice quadrata del numero così ottenuto

Dovendo ora scrivere un programma che calcoli media e scarto quadratico medio di un insieme di valori, ad esempio i voti di un

compito in classe, dovrebbe essere evidente la necessità di usare un vettore, in quanto **prima** è necessario calcolare la media sui valori acquisiti, e solo **dopo** è possibile procedere al calcolo dei singoli termini della sommatoria.

Listing 5.3: Calcolo dello scarto quadratico medio.

```
1  int main() {
2      const int MAX = 30;
3      float voti[MAX], media = 0, scarto = 0;
4      int n_compiti;
5      std::cout << "Inserisci il numero di compiti in classe:
        " << std::endl;
6      std::cin >> n_compiti;
7      for (int i = 0; i < n_compiti; ++i) {
8          std::cout << "Inserisci il voto del compito n." <<
       i + 1 << " :";
9          std::cin >> voti[i];
10         media += voti[i];
11     }
12     media = media / n_compiti;
13     for (int i = 0; i < n_compiti; ++i) {
14         scarto += pow(voti[i] - media, 2);
15     }
16     scarto = sqrt(scarto / n_compiti);
17     std::cout << "Sono stati inseriti " << n_compiti << "
       voti." << std::endl;
18     std::cout << "Media: " << media << std::endl;
19     std::cout << " Scarto quadratico medio: " << scarto <<
       std::endl;
20     return 0;
21 }
```

Il listato 5.3 permette di calcolare media e scarto quadratico medio di una serie di voti forniti come input dall'utente. In questo esempio si è supposto che il numero massimo[8] di voti da inserire sia 30, facendo successivamente inserire all'utente il numero effettivo di voti che verranno acquisiti.

Durante l'acquisizione dei valori (righe 7-11) è possibile contestualmente sommare i valori per essere già pronti a calcolare la media, cosa che verrà fatta alla riga 12. Non è invece possibile calcolare direttamente lo scarto, in quanto è necessaria la media, che non sarà disponibile se non alla fine dell'acquisizione.

Il calcolo dello scarto quadratico medio avviene quindi solo successivamente, dopo aver sommato i quadrati delle differenze tra ogni singolo valore e la media (righe 13-15), e poi facendo la divisione e l'estrazione di radice (riga 16).

8: La scelta della dimensione del vettore è sempre dipendente dal contesto e dalle ipotesi che si possono fare: in generale negli esercizi, laddove non definito, il valore preciso non è di fondamentale importanza, è sufficiente sia sensato.

Il problema dell'*out of range*

Dopo aver visto il funzionamento dei vettori, ci si potrebbe porre la domanda di cosa succeda nel momento in cui un indice assume un valore sbagliato, perché negativo o maggiore della dimensione fisica del vettore. Ad esempio, nel listato 5.3 viene chiesto all'utente di indicare quanti sono i voti che vorrà inserire successivamente: se l'utente inserisse un valore maggiore di 30, cosa succederebbe poi nel ciclo successivo, in cui chiaramente si cercherebbe di accedere a elementi del vettore con indici troppo grandi?

Per comprendere i problemi che questo comporta si faccia riferimento alla Figura 5.2

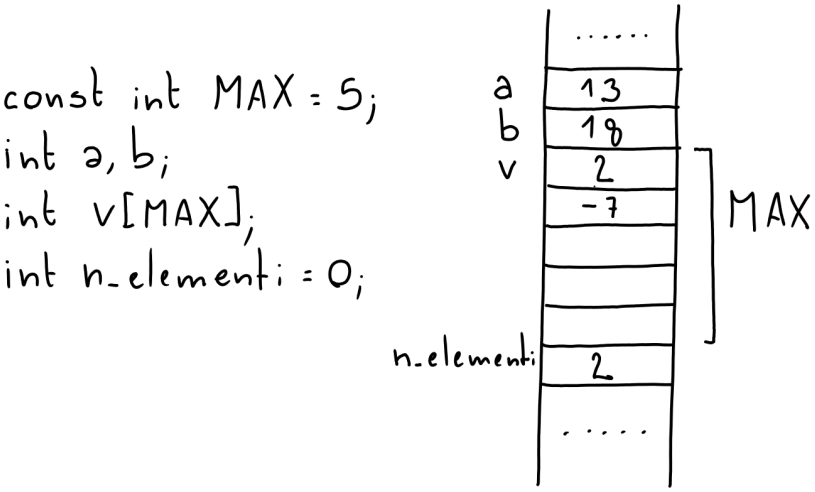


Figura 5.2.: Allocazione di un vettore in memoria

Come si può vedere in questa figura, l'insieme di celle disposte verticalmente rappresenta la memoria fisica del computer, che può essere idealizzata come una serie di celle, ognuna contenente un'informazione. Quando il programma viene tradotto, passando da codice sorgente a codice eseguibile, ad ogni variabile del linguaggio viene assegnata una porzione di memoria [9], necessaria a contenere l'informazione associata a quella variabile.

Come già detto, un vettore non è altro che un insieme di celle consecutive di memoria, quindi in Figura 5.2 si può vedere che il vettore `occupa` esattamente 5 celle ed eventuali altre variabili, in questo caso `a, b` e `n_elementi`, occupano caselle di memoria che potrebbero anche essere adiacenti[10].

Se, ad esempio, si cercasse di scrivere nel vettore all'indice 5, che non è corretto perché oltre la sua dimensione, il risultato sarebbe che il valore verrà inserito nella variabile `n_elementi`: quindi l'istruzione `v[5] = 15` potrebbe causare la sovrascrittura del valore 2 presente in `n_elementi`, facendolo diventare 15, con effetti potenzialmente disastrosi durante l'esecuzione del programma[11]. Se è chiaro il funzionamento con numeri troppo grandi, cosa succederebbe se l'indice fosse negativo?

9: L'esatta quantità di memoria dipende dal tipo della variabile, è comunque sempre un numero intero di byte, come indicato in Tabella 3.1

10: L'esatto *layout* di come le variabile vengono disposte nella memoria non è in generale sotto il controllo del programmatore, ma viene «scelto» dal compilatore, quello mostrato in figura ha solo uno scopo esemplificativo.

11: Siccome l'esatta posizione delle variabili in memoria dipende da molti fattori, l'esecuzione di questo esempio sul proprio computer difficilmente produrrà il risultato indicato qua.

Il meccanismo è il medesimo, quindi, ad esempio, un assegnamento come `v[-1] = 3` andrebbe a scrivere il valore 3 all'interno della variabile `b`, con effetti in generale imprevedibili.

Definizione di *out of range*[12]

Quando l'indice di un vettore assume un valore che si trova all'esterno dell'intervallo [0, MAX - 1], con MAX che è la dimensione del vettore al momento della dichiarazione, si sta commettendo un errore di tipo *out of range* e il comportamento del programma è *indeterminato*.

12: A seconda del linguaggio o del contesto questo problema viene anche detto *out of bounds*.

Il termine *indeterminato* indica che non è possibile prevedere con esattezza cosa potrebbe succedere. Tipicamente le situazioni che si potrebbero presentare sono le seguenti:

1. il programma sembra comportarsi in maniera corretta, ma è comunque presente un errore logico
2. il programma si comporta correttamente solo alcune volte, altre volte no
3. il programma *crasha*, ossia viene *terminato* dal S.O.

Di queste situazioni, quella di gran lunga preferibile è la terza, perché palesa la presenza di un problema, mentre le prime due possono passare inosservate anche a lungo.

5.3. La classe `vector`

I progettisti del linguaggio C++, avendo notato le problematiche nell'uso dei vettori *nativi*, hanno creato una *classe*[13] denominata `vector`, il cui scopo è offrire delle soluzioni in particolare a questi due problemi:

13: Con il termine classe si indica un particolare costrutto presente in quasi tutti i linguaggi e che verrà approfondito nel prossimo volume.

1. la dimensione fissa dei vettori, stabilita una volta per tutte all'atto della creazione
2. l'*out of range*, che non può essere eliminato, in quanto dipende da errori logici del programmatore, ma può essere gestito in modo da renderlo esplicito

Nell'esempio 5.4 si può vedere lo stesso codice già presentato nel listato 5.2, ma stavolta con l'utilizzo di `vector`, anzichè dei vettori nativi.

Listing 5.4: Esempi di input/output su un vettore.

```
1 int main() {
2     const int N = 4;
3     std::vector <int> v;
4     std::cout << "Inserisci " << N << " numeri interi." <<
      std::endl;
```

```
5     for (int i = 0; i < N; ++i) {
6         int temp;
7         std::cout << "Numero " << i + 1 << " : ";
8         std::cin >> temp;
9         v.push_back(temp);
10    }
11    //Eventuali operazioni sul vettore
12    std::cout << "Mostra a video i numeri inseriti" << std
      ::endl;
13    for (int i = 0; i < v.size(); ++i) {
14        std::cout << "Numero " << i + 1 << " : " << v.at(i)
       << std::endl;
15    }
16    return 0;
17 }
```

La prima differenza si può notare alla riga 3, dove appare la dichiarazione di un **vector**: il tipo compreso tra i simboli < e > indica cosa sono gli elementi che verranno inseriti nel vettore, in questo caso degli interi. Se si volesse avere un **vector** di **float**. la dichiarazione sarebbe la medesima, solo con **float** al posto di **int**.

Dopo la riga 3, `v` è un vettore di interi, ma al momento non ne contiene nemmeno uno, quindi la sua dimensione è 0. Nel seguito del programma vengono inseriti 4 valori e, tramite il *metodo* `push_back`, vengono *accodati* al vettore, che si "allargherà" a sufficienza per poterli contenere.

Senza entrare nei dettagli, un *metodo* è una speciale funzione, che per essere utilizzata ha bisogno di essere applicata a un oggetto, una variabile, del tipo giusto, in questo caso un **vector**. L'istruzione

```
v.push_back(temp)
```

potrebbe essere tradotta con la frase in italiano

```
accoda alla fine del vettore v il contenuto
della variabile temp
```

Il programmatore non è quindi più obbligato a dover inizialmente dichiarare la lunghezza del vettore, poiché questo sarà in grado di ridimensionarsi in modo da contenere tutti i valori che in seguito si dovesse aver la necessità di aggiungere[14].

Alla riga 14 si può invece notare come avvenga l'accesso al singolo elemento del vettore, attraverso l'utilizzo del metodo `at`, che, in un certo senso, è l'equivalente dell'utilizzo delle parentesi quadre. La differenza rispetto a queste ultime risiede nel fatto che, se si cercasse di accedere a un elemento *out of range*, il programma verrebbe terminato con un messaggio di questo tipo:

14: Il funzionamento e le implicazioni di questo comportamento verranno approfondite nel prossimo volume.

```
terminate called after throwing an instance of
'std::out_of_range'
what():  vector::_M_range_check: __n
(which is 5) >= this->size() (which is 4)
```

che indica che si è tentato di accedere all'elemento di indice 5, ma si è commesso un errore, in quanto la dimensione corrente del vettore in questione era 4 (quindi il valore massimo possibile era 3). É importante capire che, sebbene un `vector` possa ingrandirsi a piacimento durante l'esecuzione del programma, questo avviene solo a fronte della chiamata a una `push_back`.

Si può quindi affermare che il metodo `at` è equivalente all'utilizzo delle parentesi graffe, solo che in più aggiunge un controllo di correttezza sull'uscita dal range di validità dell'indice. Per retrocompatibilità le parentesi quadre possono essere utilizzate anche con i `vector`[15], rinunciando al controllo di correttezza e quindi avendo gli stessi problemi presenti nei vettori nativi.

15: Il motivo per cui i progettisti hanno deciso di permetterne l'uso ha a che fare con ragioni di efficienza, che non verranno qui approfondite.

Infine, nel listato 5.4 alla riga 13, si può vedere che la condizione di terminazione del ciclo utilizza il metodo `size` sempre applicato alla variabile `v`: questo metodo, ogni volta che viene chiamato, ritorna il numero di elementi presenti in quel momento nel vettore e, quindi, può essere usato per saper fino a che valore potrà arrivare l'indice.

Tabella 5.1.: Operazioni di base sui vettori

Operazioni	**Vettori nativi**	`vector`
Dichiarazione	`int v[MAX];`	`std::vector <int> v;`
Lettura	`int a = v[i];`	`int a = v.at(i);`
Scrittura	`v[i] = 42;`	`v.at(i) = 42;`
Dimensione	É necessario tenere memoria del numero di elementi	`v.size();`

Metodi principali della classe `vector`

Oltre ai metodi indicati in precedenza, la classe `vector` espone una serie piuttosto ampia di altri metodi che aiutano i programmatori a gestire in modo semplice le operazioni più comuni che normalmente coinvolgono i vettori. In Figura 5.3 vengono rappresentati i metodi principali, quali:

- `pop_back`: rimuove l'ultimo elemento di un vettore, può essere pensato come il complementare del metodo `push_back`. Non ha nessun parametro ed è un'operazione efficiente, allo stesso modo di `push_back`

- `erase`: rimuove l'elemento in una qualsiasi posizione valida. Da notare che la posizione non viene indicata come parametro intero, ma, per motivi che non verranno approfonditi, necessita di un *iteratore*. Un iteratore può essere pensato come una «freccia» che indica un elemento di un vettore. Per spostare questa freccia alla posizione che interessa è sufficiente aggiungere di quanti elementi deve essere spostata in avanti. Nell'esempio in figura viene utilizzato il metodo `begin` per ottenere la freccia al primo elemento del vettore e, aggiungendo il valore 1, viene spostata all'elemento in seconda posizione e quindi verrebbe eliminato il 24. Dopo l'eliminazione, tutti gli elementi successivi vengono spostati indietro di una posizione e la dimensione del vettore diminuisce di 1. Questa operazione è inefficiente, poichè richiede lo spostamento all'indietro di tutti gli elementi successivi a quello da eliminare, per «coprire il buco» lasciato dall'elemento eliminato
- `insert`: inserisce un elemento in una posizione valida. Valgono le considerazioni già fatte in precedenza sull'iteratore, quindi, nell'esempio in figura, verrebbe inserito il valore 15 nella posizione di indice 3. Anche questo metodo, come `erase`, non è efficiente, poichè tutti gli elementi successivi al nuovo elemento, devono essere «spostati» in avanti per «creare spazio» per il nuovo elemento
- `end`: è un iteratore che restituisce una freccia alla prima posizione non valida del vettore, **non** all'ultimo elemento.

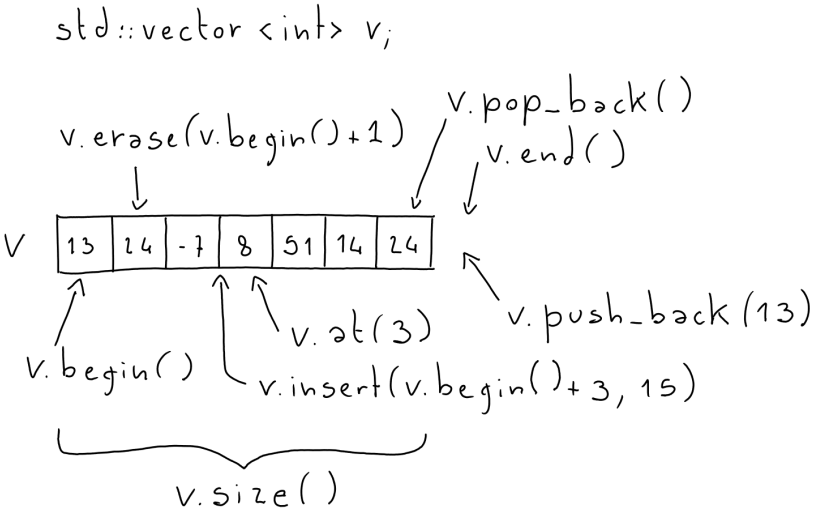


Figura 5.3.: Metodi principali della classe vector

5.4. Vettori e funzioni

Come visto nel capitolo Capitolo 4, l'organizzazione del codice attraverso la scomposizione in sottoprogrammi aiuta la gestione di

progetti complessi, permettendo l'isolamento delle funzionalità e il riutilizzo delle stesse.

Anche nel caso dei vettori, è quindi evidente che poter raggruppare le righe di codice che risolvono un particolare problema all'interno di una funzione permette di organizzare meglio i propri programmi. Nell'esempio 5.3 venivano risolti due problemi, il calcolo della media degli elementi di un vettore e lo scarto quadratico medio, si vedrà adesso come riorganizzare il codice creando due opportune funzioni, e lo si vedrà prima per i vettori nativi e, successivamente, per i `vector`.

I vettori nativi e le funzioni

L'unica differenza, rispetto a quanto già visto, nel voler utilizzare le funzioni per elaborare in un qualche modo dei vettori, risiede nel passaggio dei parametri e nell'eventuale tipo del valore di ritorno.

Volendo passare un vettore nativo[16] a una funzione è necessario infatti indicare nella dichiarazione che la variabile è appunto un vettore: per far questo verrà indicato tipo e nome del parametro seguito da una coppia di parentesi quadre[17], in maniera simile alla dichiarazione, ma senza il valore costante tra le parentesi, che invece è presente nella dichiarazione. Il concetto sottostante è che, al momento della chiamata, il parametro verrà sostituito dall'indirizzo di partenza del vettore, permettendo così alla funzione di potersi muovere tra i vari elementi ricorrendo al solito indice.

In questo modo però alla funzione manca l'informazione su quanti elementi contiene il vettore e quindi, oltre a passare il vettore come parametro, verrà passato un ulteriore intero che contiene il numero di elementi del vettore.

Per chiarire quanto esposto si guardi il codice presente nel listato 5.5:

16: Il discorso può essere esteso a un numero arbitrario di vettori passati come parametri.

17: Questa non è l'unica notazione che permette di passare dei vettori a una funzione: ne esiste una alternativa che utilizza il simbolo *, ma in questo testo è stato scelto di utilizzare le parentesi quadre perché veicolano meglio l'idea che si stia passando un vettore.

Listing 5.5: Calcolo dello scarto quadratico medio usando delle funzioni.

```
1  float media(float v[], int n) {
2      float m = 0;
3      for (int i = 0; i < n; ++i) {
4          m += v[i];
5      }
6      return m / n;
7  }
8
9  float scarto_quadratico(float v[], int n) {
10     float s = 0, m = media(v, n);
11     for (int i = 0; i < n; ++i) {
12         s += pow(v[i] - m, 2);
13     }
```

```
14      return sqrt(s / n);
15  }
16
17  int main() {
18      const int MAX = 30;
19      float voti[MAX];
20      int n_compiti;
21      std::cout << "Inserisci il numero di compiti in classe:
         " << std::endl;
22      std::cin >> n_compiti;
23      for (int i = 0; i < n_compiti; ++i) {
24          std::cout << "Inserisci il voto del compito n."
25              << i + 1 << " :";
26          std::cin >> voti[i];
27      }
28      std::cout << "Sono stati inseriti " << n_compiti
29          << " voti." << std::endl;
30      std::cout << "Media: " << media(voti, n_compiti)
31          << std::endl;
32      std::cout << "Scarto quadratico medio: "
33          << scarto_quadratico(voti, n_compiti)
34          << std::endl;
35      return 0;
36  }
```

A differenza del listato 5.3, qua vengono definite due funzioni, `media` alla riga 1 e `scarto_quadratico` alla riga 9: entrambe hanno come parametri il vettore `v` e il suo numero di elementi `n` ed entrambe restituiscono come valore di ritorno un **`float`**, che è appunto il il risultato del calcolo effettuato al loro interno.

A questo punto il loro utilizzo è del tutto simile a quello visto nel capitolo precedente, in quanto possono essere chiamate all'interno di una qualsiasi altra funzione o nel `main`. In effetti, la funzione `media` viene chiamata sia all'interno della funzione `scarto_quadratico` che del `main`: come già visto, il nome dei parametri attuali e dei parametri formali non deve necessariamente coincidere, quello che ovviamente deve coincidere è il tipo e l'ordine dei parametri che vengono passati. Va infine notato che al momento della chiamati i vettori vengono passati scrivendone il nome privato delle parentesi quadre: questo perché un vettore senza le parentesi viene interpretato come il suo indirizzo di partenza e quindi risponde a ciò che si attende la funzione.

Ultime due osservazioni importanti:

1. pur essendo possibile ritornare un vettore, questa possibilità non verrà affrontata, in quanto porterebbe a una serie di complicazioni legate alla visibilità e alla durata della variabile ritornata.
2. siccome passare un vettore vuol dire passare il suo indirizzo in memoria, ne consegue che qualsiasi modifica fatta ai

singoli elementi di un vettore, andrà a modificare gli elementi del vettore passato alla chiamata della funzione (per un esempio si veda il listato 5.6 e la Sezione 5.5). Questa proprietà del linguaggio torna inoltre utile perché l'esigenza di *ritornare* un vettore, che è stato detto che non verrà affrontata, viene di fatto compensata poiché si possono modificare i valori del vettore passato, ottenendo quindi un effetto simile.

Il seguente codice illustra come un vettore passato come parametro possa essere modificato nel corpo di una funzione: alla sua esecuzione questo codice stamperà la sequenza `2 3 4 5 6`, in conseguenza all'applicazione della funzione `incrementa` al vettore `v`, i cui elementi risultano quindi modificati.

Listing 5.6: Funzione che incrementa di uno il valore di tutti gli elementi contenuti in un vettore.

```
void incrementa(int v[], int n) {
    for (int i = 0; i < n; ++i) {
        v[i]++;
    }
}

int main() {
    int v[] = {1, 2, 3, 4, 5};
    incrementa(v, 5);
    for (int i = 0; i < 5; ++i) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

I `vector` e le funzioni

Il passaggio di un `vector` a una funzione è simile a quello già visto per il passaggio di variabili *scalari*, come è stato approfondito nel paragrafo Sezione 4.3.

Si ricorda che in C++ i due meccanismi di passaggio sono *per copia* o *per riferimento*. Mentre per i vettori nativi l'unico modo che è stato visto è quello per passaggio per copia, anche se di un indirizzo, per i `vector` possono essere usate entrambe le modalità, ma, per le considerazioni che seguono, verrà usato solo il passaggio per riferimento.

Passare un `vector` a una funzione prevede una sintassi del tutto simile a quella di una dichiarazione, con qualche differenza, come si può vedere nel codice 5.7.

Listing 5.7: Calcolo dello scarto quadratico medio usando delle funzioni e dei `vector`.

```
    float media(const std::vector <float> &v) {
        float m = 0;
        for (int i = 0; i < v.size(); ++i) {
            m += v.at(i);
```

```
5         }
6         return m / v.size();
7     }
8
9     float scarto_quadratico(const std::vector<float> &v) {
10        float s = 0, m = media(v);
11        for (int i = 0; i < v.size(); ++i) {
12            s += pow(v.at(i) - m, 2);
13        }
14        return sqrt(s / v.size());
15    }
16
17    int main() {
18        std::vector<float> voti;
19        int n_compiti;
20        std::cout << "Inserisci il numero di compiti in
      classe: " << std::endl;
21        std::cin >> n_compiti;
22        for (int i = 0; i < n_compiti; ++i) {
23            int temp;
24            std::cout << "Inserisci il voto del compito n."
       << i + 1 << " :";
25            std::cin >> temp;
26            voti.push_back(temp);
27        }
28        std::cout << "Sono stati inseriti " << n_compiti <<
       " voti." << std::endl;
29        std::cout << "Media: " << media(voti) << std::endl;
30        std::cout << "Scarto quadratico medio: " <<
      scarto_quadratico(voti) << std::endl;
31        return 0;
32    }
```

Rispetto alle funzioni viste per i vettori nativi, i parametri hanno le seguenti differenze:

- non è più necessaria la presenza del parametro che indica il numero di elementi presenti nel vettore, in quanto i `vector` «sanno» quanti elementi contengono, informazione che può essere recuperata con l'utilizzo del metodo `size`
- la dichiarazione del `vector` è quella già presentata in precedenza, solo che viene aggiunto l'operatore `&` per indicare il passaggio per riferimento. Il passaggio per riferimento dei vettori viene fatto per evitare il costo della copia, che altrimenti verrebbe effettuata elemento per elemento
- il passaggio per riferimento avviene però con due possibili modalità: costante e non costante
    - **costante**: viene preposta la parola chiave `const`, come si può vedere sia nella funzione `media` che in `scarto_quadratico` . Questo permette di evitare che il vettore venga inavvertitamente alterato all'interno della funzione e quindi

è una modalità che deve essere usata quando la funzione dove solo «leggere» gli elementi del vettore senza modificarli, ottenendo il disaccoppiamento che si ha con il passaggio per copia, senza però magari il costo che si avrebbe con quel tipo di passaggio

- **non costante**: in questo caso si utilizza un passaggio per riferimento «classico» e gli elementi del vettore saranno quindi modificabili, come si può vedere nella funzione `incrementa` del listato 5.8

la scelta tra una modalità e l'altra viene decisa dal programmatore in base al fatto che il vettore debba essere solo letto (costante) oppure modificato (non costante).

Listing 5.8: Funzione che incrementa di uno il valore di tutti gli elementi contenuti in un **`vector`**.

```
1  void incrementa(std::vector<int> &v) {
2      for (int i = 0; i < v.size(); ++i) {
3          v.at(i)++;
4      }
5  }
6
7  int main() {
8      std::vector<int> v = {1, 2, 3, 4, 5};
9      incrementa(v);
10     for (int i = 0; i < 5; ++i) {
11         std::cout << v.at(i) << " ";
12     }
13     std::cout << std::endl;
14     return 0;
15 }
```

Per quanto detto in precedenza dovrebbe risultare chiaro che il passaggio per copia di un **`vector`** va evitato.

5.5. Un esempio: inserimento e cancellazione di elementi

Come già visto, il modo più efficiente di aggiungere o rimuovere elementi da un vettore è quello di lavorare «in coda», cioè aggiungere un elemento dopo l'ultimo valido o rimuovere l'ultimo elemento: si è infatti visto che la classe **`vector`** fornisce due metodi appositamente per queste esigenze, `push_back` e `pop_back`. Potrebbe sorgere però l'esigenza di aggiungere o rimuovere un elemento da una posizione specifica del vettore: la classe **`vector`**, come già visto, dispone dei due metodi, `insert` e `erase`[18] che soddisfano questa esigenza, ma come fare per i vettori nativi?

Sia l'inserimento che la cancellazione devono preservare l'ordinamento degli elementi, quindi entrambe le operazioni comporteran-

18: Questi due metodi sono implementati allo stesso modo che si verrà in questo paragrafo per i vettori nativi, con il vantaggio di non richiedere la scrittura di codice allo sviluppare, con caratteristiche di prestazioni che sono le stesse dei vettori nativi.

no uno spostamento degli elementi, come si può vedere in Figura 5.4 e in Figura 5.5.

Volendo implementare queste operazioni utilizzando delle funzioni, per l'inserimento un possibile codice sarà il seguente:

Listing 5.9: Inserimento di un elemento in una posizione specifica.

```
int insert(int v[], int n, int pos, int el){
    for (int i = n; i > pos; --i) {
        v[i] = v[i - 1];
    }
    v[pos] = el;
    return n + 1;
}
```

Come si può notare, i parametri passati saranno il vettore `v`, la sua lunghezza logica `n`, la posizione dove si vuole inserire il nuovo elemento, indicata con `pos` e infine il valore dell'elemento da inserire. Si è poi scelto di ritornare come valore la nuova dimensione logica del vettore, in modo che il chiamante possa usare questo valore come nuova dimensione del vettore dopo l'inserimento. Questa funzione non prevede il controllo di correttezza della posizione `pos`, quindi valori al di fuori del range di validità porterebbero a comportamenti indeterminati: si lascia al lettore l'implementazione di opportuni controlli che evitino queste situazioni.

Va infine osservato che il ciclo `for` presente nella funzione parte dall'ultima posizione e procede allo spostamento degli elementi al «contrario»: anche in questo caso si lascia al lettore il compito di riflettere sul perchè questo sia necessario.

Per quanto riguarda invece la cancellazione, il codice, che risulta leggermente più semplice, è il seguente:

Listing 5.10: Cancellazione di un elemento in una posizione specifica.

```
int erase(int v[], int n, int pos){
    for (int i = pos; i < n; ++i) {
        v[i] = v[i + 1];
    }
    return n - 1;
}
```

Lo scopo del ciclo `for` è quello di spostare di una posizione all'indietro gli elementi che si trovano dopo l'elemento da eliminare, di fatto ottenendo come effetto la cancellazione dell'elemento in posizione `pos`.

Come si può vedere in Figura 5.5, la casella in posizione 6 del vettore conterrà ancora il valore 24, poichè non è possibile ovviamente eliminare fisicamente una cella di memoria, solo che logicamente non farà più parte del vettore, la cui nuova dimensione sarà 6: quindi qualsiasi successiva operazione sul vettore non la prenderà in considerazione.

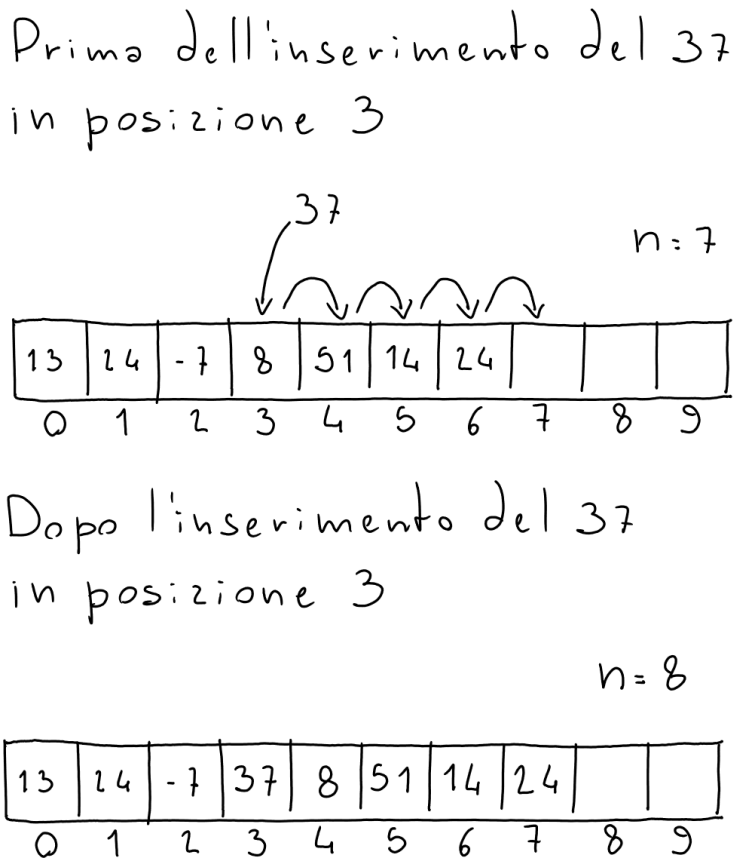


Figura 5.4.: Inserimento di un elemento in una posizione specifica

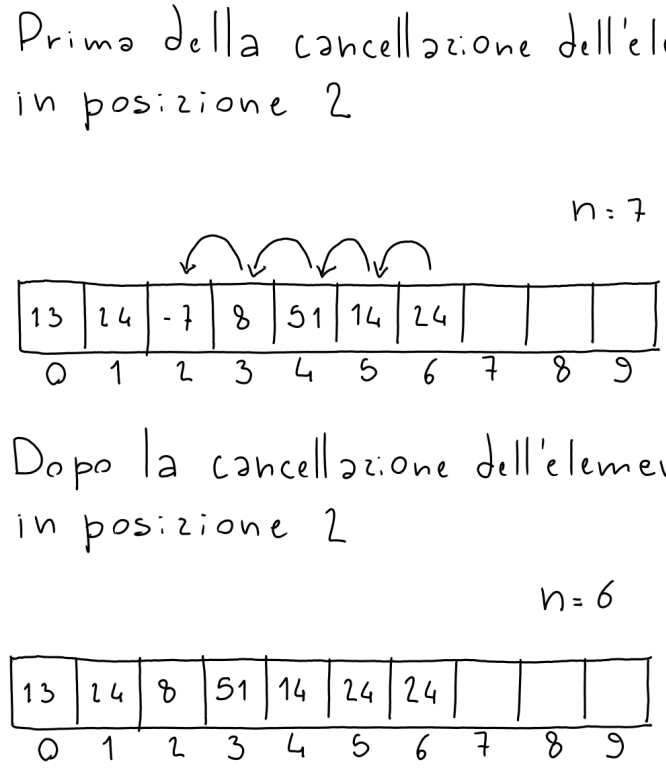


Figura 5.5.: Cancellazione di un elemento da una posizione specifica.

5.6. Le matrici

Come esistono dei problemi la cui soluzione è possibile solo attraverso l'utilizzo di vettori, esistono altri problemi la cui soluzione risulta semplificata grazie all'utilizzo di *matrici*[19].

19: In questo testo si farà riferimento esclusivamente a matrici a due dimensioni.

Definizione di matrice

Una **matrice** è l'estensione bidimensionale del concetto di vettore, dove, al posto di avere una sequenza di elementi individuati da un singolo indice, si ha una «*tabella*» in cui ogni elemento è individuato da una coppia di indici. Convenzionalmente il primo indice individua le righe e il secondo le colonne.

A differenza dei vettori, che permettono di risolvere una nuova classe di problemi in cui il numero delle variabili è arbitrario, le matrici sono solo un modo più «comodo» per affrontare dei problemi la cui rappresentazione *naturale* è già nella forma di una *griglia* o *tabella*: per fare solo alcuni esempi, si pensi al gioco della battaglia navale o del tris, a una tabella di dati in cui le righe rappresentano ad esempio degli studenti e le colonne i voti in ogni materia, o a un calendario dove le colonne sono i diversi giorni della settimana e le righe le ore della giornata[20].

20: Attenzione che, anche se la rappresentazione più naturale è in forma tabellare, non è detto che le matrici siano necessariamente gli strumenti più adatti per rappresentare i dati di un problema.

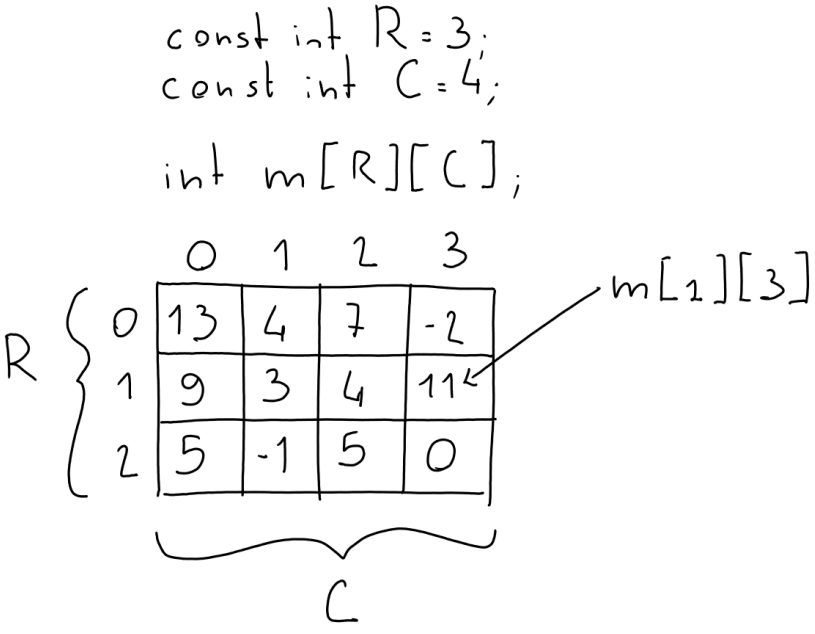


Figura 5.6.: Rappresentazione di una matrice

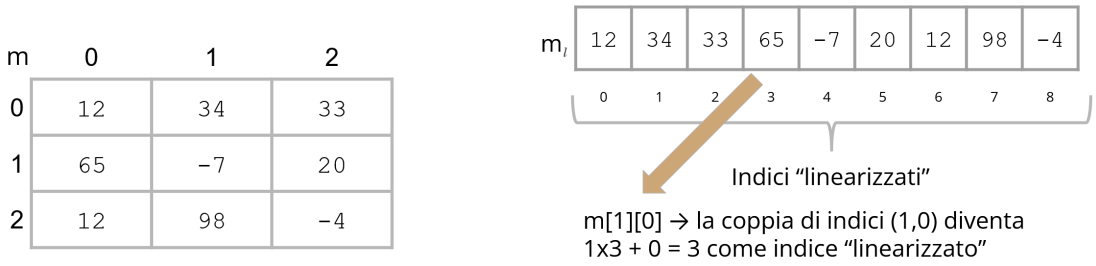
Come si può vedere in Figura 5.6, le righe e le colonne vengono indicizzate a partire da 0, quindi, per una matrice con `R` righe e `C` colonne, gli indici avranno valori compresi tra 0 e `R - 1` o `C - 1`, rispettivamente.

Sempre nell'esempio in Figura 5.6 si può vedere che la dichiarazione è simile a quella di un vettore, dove le dimensioni di riga (R) e

colonna (C) possono essere definite sempre con una dichiarazione `const`.

Ci si potrebbe domandare se, come per i vettori nativi esistono i `vector` che ne semplificano l'utilizzo, esista qualcosa di simile per le matrici. Pur potendo ricorrere all'utilizzo di `vector` di `vector`, in questo testo si utilizzeranno solo le matrici native, in quanto i problemi tipici che verranno affrontati prevederanno sempre matrici di dimensioni fisse e note a priori.

Volendo individuare un elemento, la notazione prevede di usare due coppie di parentesi quadre, dove nella prima coppia viene inserito l'indice di riga e nella seconda l'indice di colonna: nell'esempio, l'elemento `m[1][3]` individua quindi il quarto elemento della seconda riga, che contiene il valore 11.



(a) Matrice per il programmatore

(b) Matrice come compare in memoria

Figura 5.7.: Differenza tra come viene immaginata la matrice e come è memorizzata effettivamente.

Come già detto, le matrici permettono di rappresentare in modo più naturale i dati in certi tipi di problemi, ma non sono altro che uno strumento *sintattico* che utilizza come meccanismo sottostante un vettore, cosa che si può vedere in Figura 5.7. Dove a sinistra viene mostrata la matrice così come è percepita dal programmatore, a destra è mostrata la sua vera rappresentazione in memoria. Come fa il compilatore a passare da una notazione con due indici a una rappresentazione che di indice ne prevede uno solo? Viene usata la seguente formula[21]:

$$m[i][j] = m_l[i \times num\_colonne + j] \tag{5.1}$$

dove *num_colonne* è il numero di colonne della matrice che compare nella sua dichiarazione. Da questa formula si può capire perché l'elemento `m[1][0]` della matrice quadrata in Figura 5.7 è l'elemento di indice 3 nella rappresentazione interna della matrice, qui indicata con m_l, che sta per matrice linearizzata.

Sebbene le matrici siano state introdotte proprio per evitare al programmatore di dover *linearizzare* una coppia logica di indici e, quindi, di dover usare direttamente la formula, la sua conoscenza

21: La formula reale prevede l'utilizzo di indirizzi e della dimensione del tipo degli elementi del vettore, ma per la comprensione del meccanismo di traduzione, quella proposta è del tutto adeguata.

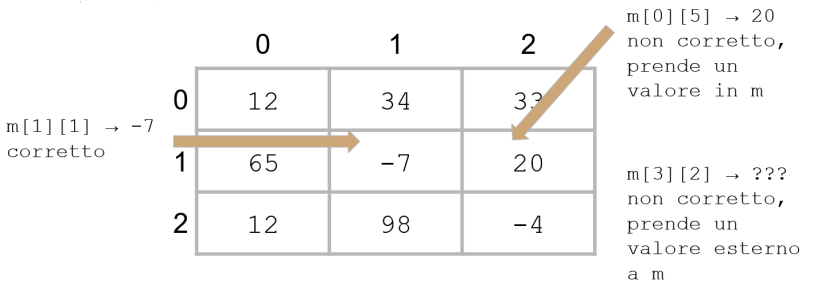


Figura 5.8.: Errori nell'utilizzo degli indici.

permette di capire il perché di alcuni comportamenti «patologici», che si possono trovare nella Figura 5.8.

Mentre l'elemento di indici (1, 1) è sicuramente corretto e corrisponde al valore -7. Se invece gli indici fossero (0, 5), sicuramente ci sarebbe un errore logico perché l'indice di colonna risulta maggiore del massimo consentito per quella matrice, che è 2: questo però non comporta un problema di *out of range*, poiché dall'Equazione 5.1, risulta che la casella selezionata è quella di indice 5, indice ottenuto come `0 * 3 + 5`, e il valore contenuto è 20.

Sempre usando la stessa formula si può invece vedere che la coppia di indici (3, 2) andrebbe a selezionare la casella di indice linearizzato 11, ottenuto come `3 * 3 + 2`, che si trova al di fuori del vettore, che ha indice al massimo 8, e quindi si avrebbe lo stesso problema di *out of range* già visto per i vettori.

5.7. Esempio: ripetizioni di un voto

Si vedrà adesso un esempio per approfondire l'utilizzo delle matrici, che affronterà anche il problema del passaggio di una matrice a una funzione.

27	19	22	30	24
30	20	28	30	19
18	22	30	26	27
30	30	25	23	30

Tabella 5.2.: Esempio dei voti di 4 appelli diversi.

Si supponga di voler realizzare un programma che permetta di inserire i voti di tutti gli appelli dell'anno di un certo esame, per poi poter rispondere a delle domande del tipo «Quanti studenti hanno preso 30 nei vari appelli?». Un modo di organizzare i dati su cui lavorerà il programma è quello di utilizzare appunto una matrice, dove ogni riga rappresenterà un particolare appello e ogni colonna un voto, come si può vedere nella tabella Tabella 5.2, dove sono presenti quattro appelli e ogni appello contiene 5 voti[22].

22: Questa è chiaramente una situazione semplificata, poiché molto difficilmente appelli diversi avranno un numero uguale di studenti partecipanti, e quindi di voti, ma il caso reale può essere facilmente gestito a patto di utilizzare un codice un po' più sofisticato.

Un possibile codice che risolve questo problema si può vedere nel listato 5.11, che eseguito potrebbe dare origine al seguente output (o altro in dipendenza dalle scelte dell'utente):

```
Inserisci il voto: 30
Appello n. 1 - studenti che hanno preso 30: 1
Appello n. 2 - studenti che hanno preso 30: 2
Appello n. 3 - studenti che hanno preso 30: 1
Appello n. 4 - studenti che hanno preso 30: 3
```

Verranno adesso approfonditi alcuni aspetti del codice, evidenziando le caratteristiche che possono essere generalizzate partendo da questo esempio.

Listing 5.11: Esempio di utilizzo di una matrice.

```
1  const int APPELLI = 4;
2  const int VOTI = 5;
3
4  void trova_uguali(int m[][VOTI], int v[],
5              int n, int voto){
6      for (int i = 0; i < n; i++)
7          v[i] = 0;
8      for (int i = 0; i < n; i++)
9          for (int j = 0; j < VOTI; j++)
10             if(m[i][j] == voto)
11                 v[i]++;
12 }
13
14 int main(void){
15     int m[][VOTI] = {
16         {27, 19, 22, 30, 24},
17         {30, 20, 28, 30, 19},
18         {18, 22, 30, 26, 27},
19         {30, 30, 25, 23, 30}
20     };
21     int v[VOTI], voto;
22     std::cout << "Inserisci il voto: ";
23     std::cin >> voto;
24     trova_uguali(m, v, APPELLI, voto);
25     for (int i = 0; i < APPELLI; i++){
26         std::cout << "Appello n. " << i + 1 <<
27         " - studenti che hanno preso " << voto << " : "
28         << v[i] << std::endl;
29     }
30     return 0;
31 }
32
```

Dichiarazione e inizializzazione di una matrice

Nel listato 5.11 la matrice viene dichiarata e contestualmente inizializzata con dei valori utilizzando lo stesso metodo visto in

Sottosezione 6 per i vettori: come si vede alla linea 15, come per i vettori, la prima coppia di parentesi non contiene nessun numero, poiché il numero di righe verrà dedotto dal compilatore in base al numero di righe che seguono, in questo esempio 4. Da notare che ogni riga è di fatto un vettore, la cui lunghezza è determinata dal numero di colonne della matrice, 5 in questo esempio.

Se la matrice non fosse inizializzata contestualmente alla dichiarazione, la sintassi, come già visto, sarebbe la seguente:

```
int m[APPELLI][VOTI];
```

sempre con l'accortezza di non usare *numeri magici*, ma preferire etichette linguistiche definite tramite variabili costanti (`const`).

Passaggio di una matrice a una funzione

Purtroppo per mantenere il codice semplice, il passaggio di una matrice a una funzione non rispetterà il requisito di indipendenza dal resto del codice, che invece era stato detto fosse un obiettivo nella progettazione delle funzioni.

Guardando infatti la funzione definita alla riga 4 del listato 5.11 si vedere che sono presenti 4 parametri:

1. la matrice, qui indicata con `m`: compaiono due coppie di parentesi quadre, la prima risulta vuota e la seconda contiene invece il numero di colonne della matrice che verrà passata. Per quanto riguarda la prima coppia, il fatto che sia vuota rispecchia la stessa situazione presente nei vettori, in cui sarà necessario poi passare il numero di righe come parametro aggiuntivo. Per la seconda, invece, inserire un numero è una condizione necessaria e può essere facilmente spiegata ricordando la formula **??**: per poter calcolare l'indirizzo di una qualsiasi cella della matrice individuata da due indici è necessario conoscere il numero di colonne[23] della matrice (*num_colonne* nella formula), per permettere al compilatore di *linearizzare* la notazione e poter accedere alla locazione di memoria corretta. Questa situazione implica che la funzione sarà specifica per matrici con quel particolare numero di colonne. portando al problema prima accennato di una notazione che impedisce alla funzione di poter essere generale e indipendente da elementi esterni (in questo caso `VOTI`)
2. il vettore `v[]`, che servirà a contenere il numero di voti di ogni appello che soddisfano la condizione di essere uguali a `voto`, passato come quarto parametro. In questo caso la sintassi di passaggio è quella dei vettori, sarà chiaramente cura del programmatore passare un vettore che abbia un numero di

23: Se si passasse alla funzione una matrice con un numero di colonne differente da quanto dichiarato, l'esecuzione produrrebbe un risultato non corretto.

elementi sufficienti a contenere il risultato dell'elaborazione, che in questo caso è pari al numero di appelli.
3. il numero intero `n`, che rappresenta sia il numero di righe della matrice che il numero di elementi del vettore, in quanto, come già detto, devono coincidere
4. il parametro `voto`, che indicherà il voto da ricercare e contare per ogni appello, cioè per ogni riga della matrice.

Una volta indicati correttamente i parametri, la funzione inizializzerà a zero tutti gli elementi del vettore `v`, in quanto può essere pensato come un vettore di *contatori* e, come tale, richiede in partenza che contengano il valore 0. Successivamente i due cicli `for` annidati esplorano tutta la matrice riga per riga e, per ogni elemento, verificano se è uguale al `voto` richiesto. In caso affermativo viene incrementato il contatore corrispondente alla riga *i*-esima.

Il doppio ciclo `for` è spesso associato alla presenza di una matrice, ma non deve essere confuso con una condizione necessaria: ci sono difatti elaborazioni di matrici che possono richiedere un solo ciclo `for` o altre che ne richiedono tre o più.

Chiamata di una funzione che accetta matrici

Per completare, si può vedere nel `main` che, dopo aver acquisito la scelta del voto da parte dell'utente, viene chiamata la funzione `trova_uguali`, analizzata in precedenza. Come si può vedere la matrice viene passata utilizzando solo il suo nome, `m`, senza nessuna parentesi, allo stesso modo in cui avveniva per i vettori, come si può vedere anche per il passaggio di `v`.

5.8. Esercizi

Studi sui vettori

Ognuno di questi esercizi, con eventuali piccole modifiche dovute alle differenze che li caratterizzano, può essere svolto utilizzando i vettori nativi o i `vector`.

1. Dichiarare un vettore di 10 elementi interi, chiedere all'utente quanti valori vuole inserire (massimo 10), fargli inserire quei valori e successivamente ristampare su una riga il numero di elementi pari presenti e successivamente il loro elenco e sulla riga successiva il numero di elementi dispari e successivamente il loro elenco.
2. Dichiarare un vettore di interi di 10 elementi, riempirlo con valori casuali compresi tra *a* e *b*, con *a* e *b* inseriti dall'utente, successivamente stampare su una riga quelli compresi tra *a* e *m* e sulla riga successiva quelli compresi tra *m+1* e *b*, con **m=(a+b)/2**.
3. Dichiarare un vettore di 10 elementi, riempirlo con valori casuali qualsiasi, e stamparlo in orizzontale in modo che nella prima riga compaiano gli indici e nella seconda i valori corrispondenti, come in questo esempio

0	1	2	3	4	5	6	7	8	9
12	54	34	87	7	88	66	49	82	33

4. Dichiarare un vettore di 10 elementi, facendo inserire successivamente dei valori interi fino a quando non viene inserito un valore negativo oppure sono stati inseriti 10 numeri. Successivamente ristampare i numeri inseriti in ordine inverso.
5. Dichiarare un vettore di 50 elementi, chiedere all'utente di inserire un valore N positivo < 50 e fare inserire N elementi nel vettore. Successivamente invertire l'ordine degli elementi all'interno dal vettore, in modo che il primo inserito vada in ultima posizione, quello in seconda vada in penultima ecc. ecc. e successivamente stamparlo.
6. Dichiarare un vettore di 20 elementi reali, inserire 20 valori reali ottenuti casualmente e compresi tra 0 e 100 con almeno una cifra decimale. Successivamente chiedere all'utente di inserire due valori *a* e *b* tali che $0 \leq a < b < 20$ e stampare la somma degli elementi compresi tra l'indice *a* e l'indice *b*. Continuare a ripetere la richiesta della coppia di indici all'utente e calcolare la somma, fino a quando non viene inserito un indice negativo.
7. Dichiarare due vettori, il primo composto da 10 elementi e il secondo da 5. Dopo aver fatto inserire tutti i valori nei due vettori, verificare se il secondo vettore è contenuto nel primo, cioè se tutti gli elementi del secondo sono contenuti anche nel primo. Stampare quindi se il secondo vettore è contenuto nel primo oppure no.
8. Dichiarare due vettori, il primo composto da 10 elementi e il secondo pure. Dopo aver fatto inserire tutti i valori nei due vettori, calcolare l'unione dei due. L'unione di due insiemi è l'insieme degli elementi contenuti in almeno uno dei due vettori, eventualmente preso una volta soltanto.
9. Dichiarare due vettori, il primo composto da 10 elementi e il secondo pure. Dopo aver fatto inserire tutti i valori nei due vettori, calcolare l'intersezione dei due. L'intersezione di due insiemi è l'insieme degli elementi contenuti in entrambi i vettori.
10. Dichiarare un vettore di 20 elementi interi, inserire successivamente 10 valori ottenuti casualmente e compresi tra 0 e 100. I valori devono essere messi nelle prime 10 posizioni e le altre dieci saranno quindi "libere". Successivamente chiedere all'utente un valore e la

posizione dove lo si vuole inserire. I valori nelle posizioni successive a quella di inserimento devono essere spostati tutti avanti di una posizione, in modo da non perdere valori. Il programma deve continuare a inserire elementi fino a quando non c'è più posto nel vettore per inserire nuovi elementi. Dopo ogni inserimento si stampino gli elementi del vettore per verificare che l'inserimento è andato a buon fine.

11. Dichiarare un vettore di 20 elementi interi, inserire 20 valori ottenuti casualmente e compresi tra 0 e 100. Successivamente chiedere all'utente la posizione di un elemento che si vuole eliminare e lo si elimini dal vettore in modo che i numeri che si trovano dopo quello da eliminare vengano spostati ognuno alla posizione precedente. Il programma deve continuare a chiedere la posizione di un elemento da eliminare fino a quando l'utente non inserisce una posizione non valida. Dopo ogni eliminazione si stampino gli elementi del vettore per verificare che l'eliminazione è andata a buon fine.
12. Scrivere un programma che, dati in input **n** numeri reali, con **n** che al massimo vale 100, stampi *quanti* di essi sono maggiori della media, stampandoli poi a video.
13. Scrivere un programma che generi 100 numeri interi casuali compresi tra 0 e N, estremi inclusi, con N intero positivo o negativo (ma non zero) e che stampi **quante** coppie tra gli elementi generati danno come somma N.

Studi sulle matrici

1. Si definisca una matrice NxN, con N a scelta, si mettano a zero tutte le sue celle, si chieda all'utente di inserire un valore qualsiasi in una cella individuata dall'indice di riga e colonna, si stampi la matrice e si ripeta il procedimento fino a quando l'utente non inserisce il valore -1. L'utente può anche inserire il valore in una cella già usata in precedenza, nel qual caso si osservi che il valore precedente viene sovrascritto dal nuovo valore.
2. Si definisca una matrice NxM, con N e M a scelta e si riempiano tutte le caselle con numeri casuali compresi tra 0 e 100. Si chieda poi all'utente di inserire un indice di riga o di colonna a propria scelta e il programma stampi i valori contenuti nella riga o nella colonna scelta e la loro somma. Il procedimento di scelta fatto dall'utente deve essere ripetuto fino a quando lo desidera.
3. Si definisca una matrice NxN, con N a scelta e si riempiano tutte le caselle con numeri casuali compresi tra 0 e 100. Si stampi successivamente il valore della somma delle celle appartenenti alla *diagonale principale* e la somma di quelli appartenenti alla *diagonale secondaria*. La diagonale principale è quella composta dalle celle che partono dall'angolo in alto a sinistra fino ad arrivare a quello in basso a destra, mentre la diagonale secondaria è quella che parte dall'angolo in alto a destra fino ad arrivare a quello in basso a sinistra.

Tabella 5.4.: La diagonale corrispondente a R = 1 è evidenziata in giallo (somma = 54), la diagonale corrispondente a C = 2 è evidenziata in rosa (somma = 92).

22	56	21	6	14
12	13	73	8	17
44	23	54	61	63
33	3	19	30	41

4. Si definisca una matrice NxM, con N ed M a scelta e si riempiano tutte le caselle con numeri casuali compresi tra 0 e 100. Si chieda all'utente di inserire un numero di riga R o di colonna C e si stampi successivamente il valore della somma delle celle appartenenti alla *diagonale*

individuata dall'indice di riga R o di colonna C. La diagonale individuata da un indice di riga è quella che parte dalla prima cella della riga R fino ad arrivare all'ultima riga o colonna, scendendo in diagonale, mentre la diagonale individuata da un indice di colonna C è quella che parte dalla prima cella della colonna C fino ad arrivare all'ultima riga o colonna, scendendo in diagonale. In tabella 5.4 si possono vedere due esempi di diagonali, individuate rispettivamente dalla riga e dalla colonna.

5. Si definisca una matrice NxN, con N a scelta e si riempiano tutte le caselle con numeri casuali compresi tra 0 e 100. Successivamente si calcolino le somme di tutte le cornici concentriche e le si stampino a video. Nella tabella 5.5 si può vedere un esempio di cosa si intenda per cornici. *Suggerimento*: provare a calcolare prima la somma per una cornice specifica, ad esempio quella più esterna e, una volta capito il meccanismo, provare a generalizzarlo per una qualsiasi cornice.

Tabella 5.5.: In questa matrice 5x5 le cornici sono 3, quella esterna di colore bianco, quella successiva di colore giallo e anche la casella singola, di colore rosa, che conta comunque come matrice

22	56	21	6	14
12	13	73	8	17
44	23	54	61	63
33	3	19	30	41
14	99	31	66	15

6. Si definisca una matrice NxM, con N e M a piacere e si scriva una funzione che abbia come parametri la matrice `m`, un intero `n` che rappresenti un indice di riga o di colonna e un ultimo parametro `q` che indica se il parametro precedente è un indice di riga, se vale 1, o di colonna, se vale 2. La funzione deve calcolare la somma degli elementi della riga o colonna `n`, in base al valore di `q` e ritornare la somma calcolata. Si può supporre che sia `n` che `q` siano sempre valori validi.
   Successivamente si scriva un programma che permetta di verificare la correttezza della funzione scritta.
7. Scrivere una funzione che abbia come parametri una matrice `m` di dimensioni N x N e un vettore `v` di lunghezza N e che ritorni il totale di righe e/o colonne che sono uguali al vettore passato (si veda l'esempio nella tabella 5.6).
   Successivamente si scriva un programma che permetta di verificare la correttezza della funzione scritta.

Tabella 5.6.: In questo esempio la funzione dovrebbe ritornare 2, poichè ci sono 1 riga (la prima) e una colonna (l'ultima), il cui contenuto è uguale a quello del vettore v.

`m`

1	2	1
3	7	2
4	5	1

`v`

1	2	1

Esercizi sui vettori

1. Il crivello di Eratostene è un antico procedimento per il calcolo delle tabelle di numeri primi fino ad un certo numero **n** prefissato. Il procedimento è il seguente: si scrivono tutti i numeri

naturali a partire da 2 fino n in un elenco detto setaccio. Poi si cancellano (setacciano) tutti i multipli del primo numero del setaccio (escluso lui stesso). Si prende poi il primo numero non cancellato maggiore di 2 e si ripete l'operazione con i numeri che seguono, proseguendo fino a che non si applica l'operazione all'ultimo numero non cancellato. I numeri che restano sono i numeri primi minori o uguali a n. Scrivere un programma che chieda all'utente di inserire un numero **n** e che, utilizzando il procedimento esposto in precedenza, stampi l'elenco di tutti i numeri primi compresi tra 2 e **n**.

2. Scrivere un programma che, dati in input **n** numeri reali, con **n** che al massimo vale 100, esegua un filtraggio a *media mobile*, cioè sostituisca ogni elemento in posizione *i* con la media ottenuta prendendo i **k** elementi precedenti, l'elemento stesso e i **k** elementi successivi. Per gli elementi che si trovano nelle prime **k** posizioni e nelle ultime **k** posizioni, la media va fatta solo con gli elementi validi.
3. Si scriva un programma che inserisca 100 interi casuali compresi tra 10 e 1100, estremi inclusi, all'interno di un vettore A e successivamente faccia la stessa cosa per un secondo vettore B. Nell'inserimento dei numeri casuali bisogna garantire che né A né B abbiano duplicati al loro interno, mentre invece potrebbero esserci numeri che compaiono sia in A che in B. Successivamente il programma dovrà copiare in un terzo vettore C l'*intersezione* di questi due vettori, dove l'**intersezione** di due insiemi è l'insieme di tutti i valori che sono compresi in entrambi gli insiemi a cui viene applicata. Infine il programma dovrà stampare i valori contenuti nel vettore C.
4. Scrivere un programma che faccia inserire all'utente due numeri interi **N** e **K** (si può supporre che $1 < K < N$ e che N sia multiplo di K). A questo punto generare una sequenza di N numeri casuali compresi tra 1 e 100 (estremi inclusi) e stampare quanto vale il massimo tra tutte le somme degli interi appartenenti a un insieme di numeri *K-distanziati*. Un insieme di numeri **K-distanziati** all'interno di una sequenza di N numeri, è formato da tutti gli elementi che si trovano a distanza di K posizioni dal precedente e dal successivo. Ad esempio, dato l'insieme degli N = 9 numeri 5, 76, 81, 18, 23, 67, 3, 14, 42 e avendo inserito K = 3, allora un insieme 3-distanziato è quello formato dai numeri 5, 18, 3, un altro insieme è quello formato da 76, 23, 14 ecc. In questo esempio il programma stamperà 190, che è la somma degli elementi 81, 67 e 42 e che è il massimo tra le somme dei vari insiemi.
5. Scrivere una funzione che accetti un vettore **v** e un intero **n** che ne rappresenta il numero di elementi, e che inverta la posizione degli elementi in esso contenuti, in modo che il primo diventi l'ultimo e viceversa, che il secondo diventi il penultimo e viceversa, e così via. La funzione non deve usare un secondo vettore, ma lavorare direttamente sul vettore da invertire. Successivamente si scriva un programma che, dopo aver scritto 10 valori casuali compresi tra 200 e 500 in un vettore, passi il vettore alla funzione e verifichi che, dopo la chiamata alla funzione, gli elementi risultino invertiti.
6. Si scriva una funzione che accetta come parametri un vettore **v** di double e un intero **n** che ne rappresenta il numero di elementi, e che ritorni la media dei valori contenuti nel vettore.
   Si scriva successivamente un programma per testare la correttezza della funzione creata.
7. Si scriva una funzione che accetti come parametri due vettori di double **a** e **b** con lo stesso numero **n** di elementi, passato come terzo parametro, e che ritorni il *prodotto scalare* tra i due. Il prodotto scalare tra due vettori è un'operazione algebrica così definita:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

   dove a_i e b_i sono gli elementi in posizione *i*-esima del vettore. Si scriva successivamente un

programma per testare la correttezza della funzione creata.

8. Scrivere un programma che, utilizzando due funzioni che simulano il lancio di uno e due dadi rispettivamente, conti il numero di occorrenze di ogni valore uscito e mostri il numero di occorrenze di ogni valore, anche in termini percentuali, come nell'esempio seguente ottenuto con 100000 lanci.

```
Distribuzione nel lancio di un dado
1: 16647 volte (16.647000%)
2: 16753 volte (16.753000%)
3: 16797 volte (16.797000%)
4: 16529 volte (16.529000%)
5: 16450 volte (16.450000%)
6: 16824 volte (16.824000%)
Distribuzione nel lancio di due dadi
2: 2776 volte (2.776000%)
3: 5494 volte (5.494000%)
4: 8333 volte (8.333000%)
5: 11214 volte (11.214000%)
6: 13899 volte (13.899000%)
7: 16695 volte (16.695000%)
8: 13941 volte (13.941000%)
9: 11087 volte (11.087000%)
10: 8343 volte (8.343000%)
11: 5511 volte (5.511000%)
12: 2707 volte (2.707000%)
```

9. Si scriva una funzione che accetti come parametri un vettore di interi **v**, un intero **n** che ne rappresenta il numero di elementi e un intero **max** che rappresenta l'altezza massima dell'*istogramma* che questa funzione dovrà disegnare a video utilizzando degli asterischi. L'istogramma è quella rappresentazione grafica che associa ad un insieme di valori una serie di rettangoli di altezza proporzionale ai valori.
   Il disegno effettuato dalla funzione dovrà essere fatto in questo modo: supponendo che **v** = 23, 34, 18, 5, 28 e **max** valga 10, allora il risultato sarà il seguente:

```
******
**********
*****
*
********
```

   cioè il valore massimo (il numero degli asterischi della riga più lunga) sarà portato a **max**, cioè 10, e gli altri valori verranno proporzionati di conseguenza, arrotondando all'intero inferiore. Si scriva successivamente un programma, verifichi visivamente, attraverso l'utilizzo della funzione appena definita, l'uniformità della distribuzione dei lanci di un dado utilizzando la funzione `rand` (come nell'esercizio precedente): il programma dovrà simulare un numero grande di lanci di dadi (ad esempio 100000) e contare quante volte è uscito l'1, quante volte il 2 ecc. Questi valori sono quelli che dovranno essere passati alla funzione per disegnare

l'istogramma, che dovrebbe risultare con tutti i rettangoli della stessa altezza, o al massimo con una differenza di un asterisco dovuto all'approssimazione.
Si rifaccia la stessa cosa con il lancio di due dadi e si verifichi che l'istogramma risultante abbia una forma a "campana", cercando anche di capirne intuitivamente il perchè.
Come ultima modifica si provi a riscrivere la funzione in modo che stampi l'istogramma in verticale, come succede normalmente negli istogrammi. Nell'esempio precedentemente l'istogramma diventerebbe:

```
    *
    *
    *     *
    *     *
  * *     *
  * * *   *
  * * *   *
  * * *   *
  * * *   *
  * * * * *
```

10. Una ditta di telecomunicazioni ha installato una linea formata da ponti radio, ognuno dei quali può ricevere o trasmettere un segnale solo dal ponte radio adiacente (si supponga che i ponti radio siano disposti lungo una linea retta che parte da **0** e arriva a **N** Km). Grazie a delle nuove tecnologie che consentono una maggiore distanza di trasmissione/ricezione, si vuole verificare se una coppia di ponti radio adiacenti possa essere sostituita da un solo ponte radio posizionato esattamente nel punto intermedio tra i due (il ponte radio iniziale e quello finale non possono essere mai sostituiti perché sono nelle posizioni da dove deve partire e arrivare la comunicazione). Scrivere una funzione che abbia come parametro un vettore contenente le posizioni ordinate dei ponti radio, il numero di elementi contenuti e un numero **P** che rappresenta la distanza massima che possono coprire i nuovi dispositivi: la funzione dovrà ritornare il numero di coppie di ponti radio che possono essere sostituite da un solo ponte radio posizionato nella posizione intermedia tra i due, in modo che la sua distanza dai ponti radio precedenti e successivi alla coppia da sostituire sia minore o uguale a P. Se ad esempio il vettore contenesse i seguenti valori []0, 34, 44, 88, 150] e la distanza P fosse 50, il valore di ritorno sarebbe 1, poichè solo la coppia 34, 44 potrebbe essere sostituita da un nuovo ponte radio piazzato in 39: infatti la sua distanza sarebbe minore di 50 sia da quello posizionato al Km 0 che da quello posizionato al Km 88. Successivamente scrivere un programma che, dopo aver fatto inserire all'utente il numero di ponti radio, la loro posizione in ordine crescente e la distanza di trasmissione/ricezione **P** dei nuovi dispositivi, stampi a video il numero di coppie candidate a essere sostituite con un nuovo ponte radio.
11. Si scriva un programma che, dopo aver fatto inserire n = 10 interi casuali con valori compresi tra 0 e 100, estremi inclusi, all'interno di un vettore, trovi la posizione del *baricentro* del vettore. Per baricentro si intende l'indice **k** del vettore tale per cui, detta **S1** la somma degli elementi di indice compreso tra 0 e K e **S2** la somma degli elementi di indice compreso tra k+1 e n-1, il valore assoluto di **S1 - S2** deve essere il minimo possibile.
12. Si scriva un programma che, dopo aver generato 100 interi casuali all'interno di un vettore con valori compresi tra 1 e 100, estremi inclusi, trovi il valore mediano dell'insieme di interi. Il valore mediano di un insieme é il valore dell'elemento che, una volta ordinato l'insieme dall'elemento

più piccolo al più grande, si trova nella posizione di indice (N - 1)/2, con N il numero di elementi dell'insieme. Ad esempio con l'insieme [7, 12, 4, 33, 21, 18] il valore dell'elemento mediano è 12. **Suggerimento**: non è necessario ordinare il vettore, basta procedere in questo modo: si cerca il massimo per $N/2$ volte e ogni volta lo si "esclude" sovrascrivendolo con -1, quindi si cerca il massimo ancora una volta e si trova il mediano.

Esercizi sulle matrici

1. Un programma di elaborazione di immagini deve essere in grado di verificare se un'immagine in scala di grigi abbia sul suo contorno una cornice nera oppure no. Si supponga che ogni immagine sia composta da una griglia quadrata di pixel di dimensioni NxN, con N al massimo di valore 1000, dove la luminosità di ogni pixel è rappresentata da un valore compreso tra 0 e 255 (0 = nero, 255 = bianco). Il programma rileva la presenza di una cornice se almeno il 90% dei pixel al bordo dell'immagine ha una luminosità minore di 10. Si scriva un programma che crei casualmente i valori dei pixel per un'immagine di dimensione NxN, con N a piacere, e che indichi se contiene una cornice oppure no. Per verificare la presenza della cornice si scriva una funzione che ricevuti come parametri una matrice, restituisca 1 se è presente una cornice, 0 altrimenti.
2. Il 2 gennaio 2019 il lander cinese Chang'e-4 è atterrato (o allunato) sulla faccia nascosta della Luna, primo oggetto creato dall'uomo a compiere questa missione. Durante la fase di discesa* il lander ha scattato molte foto a risoluzione 640x480, in ognuna della quali un singolo pixel rappresenta o uno stato di oscurità (0) o uno stato di luce (1). Per ottenere un atterraggio morbido il lander analizzava le foto, verificando se fossero presenti delle rocce e modificando quindi la propria traiettoria per ottenere un atterraggio migliore. Supponendo che una roccia all'interno di una foto sia rappresentata da un segmento orizzontale di almeno 10 pixel consecutivi tutti di valore 0, individuare quante rocce sono presenti all'interno di una foto. Per far ciò realizzare una funzione che, presi come parametri una matrice delle dimensioni di una foto, ritorni il numero di rocce in essa contenute. Successivamente si scriva un programma che crei una foto composta da pixel casuali con valori 0 o 1, chiami la funzione scritta in precedenza e stampi il numero di rocce presenti nella foto.
3. Per mantenere bilanciata una "piattaforma di trasporto carichi" sono stati inseriti dei sensori di peso su tutta la sua superficie. La piattaforma ha forma quadrata di lato 10 metri ed è divisa in una griglia "quadrettata" in cui ogni quadretto ha lato di un metro e c'è un sensore disposto al centro di ognuno di essi. Ogni sensore da una lettura che è un numero intero compreso tra 0 e 1023. Scrivere un programma che, acquisiti da tastiera tutti i valori letti dai 100 sensori, verifichi se la piattaforma è *sbilanciata*: la piattaforma è sbilanciata se la somma dei valori letti dai sensori all'interno di un quadrante è più del 30% del valore della media della somma dei valori degli altri tre quadranti. Un quadrante è una delle quattro parti in cui viene divisa la piattaforma tracciando due segmenti perpendicolari tra di loro e paralleli ai lati, passanti per il centro. Per valutare lo sbilanciamento si scriva una funzione che, avendo come parametri la matrice dei valori dei sensori, ritorni 1 se la piattaforma è sbilanciata, 0 se invece è bilanciata.
4. Il 10 aprile 2019 la collaborazione internazionale EHT ha pubblicato la prima "foto" di un buco nero, ottenuta combinando i dati di una rete di 8 radiotelescopi sparsi per il mondo†. Supponendo che una singola foto sia composta da una matrice 64x64 di numeri interi, scrivere

* Da qui in poi la narrazione diventa non del tutto reale...

† Per chi fosse interessato può trovare un primo spunto all'indirizzo `https://www.bbc.com/news/science-environment-47891902`

una funzione che combini due matrici nel seguente modo: faccia il prodotto di ogni elemento della prima matrice $a_{i,j}$ per l'elemento $b_{i,j}$ della *trasporta* della seconda e il risultato lo inserisca nella posizione *i,j* della prima matrice. La trasposta di una matrice è la stessa matrice in cui la prima riga diventa la prima colonna, la seconda riga diventa la seconda colonna etc.

Progetti

Istogramma in grafica

Prendendo spunto dall'esercizio della sezione 5.8, si crei un programma grafico che mostri l'istogramma di una serie di valori, in modo che:

- l'istogramma occupi tutta l'area di disegno indipendentemente dal numero di valori da inserire nell'istogramma e dal loro valore
- i valori dei singoli rettangoli siano rappresentati in modo opportuno all'interno del disegno
- (opzionale) i valori dell'istogramma possano essere letti da un file anzichè inseriti nel codice

Il programma, inoltre, deve permettere, tramite la pressione di appositi tasti:

- di disegnare l'istogramma in orizzontale o in verticale
- di disegnare ogni rettangolo dello stesso colore oppure di colori differenti

Filtraggio a media mobile

Prendendo spunto dall'esercizio della sezione 5.8, si crei un programma grafico che mostri l'andamento di una serie di valori e la versione degli stessi valori filtrati utilizzando la media mobile, disegnati in un piano cartesiano in due diversi colori e con l'indicazione dei valori sulle ascisse e sulle ordinate, come nell'esempio della Figura 5.9

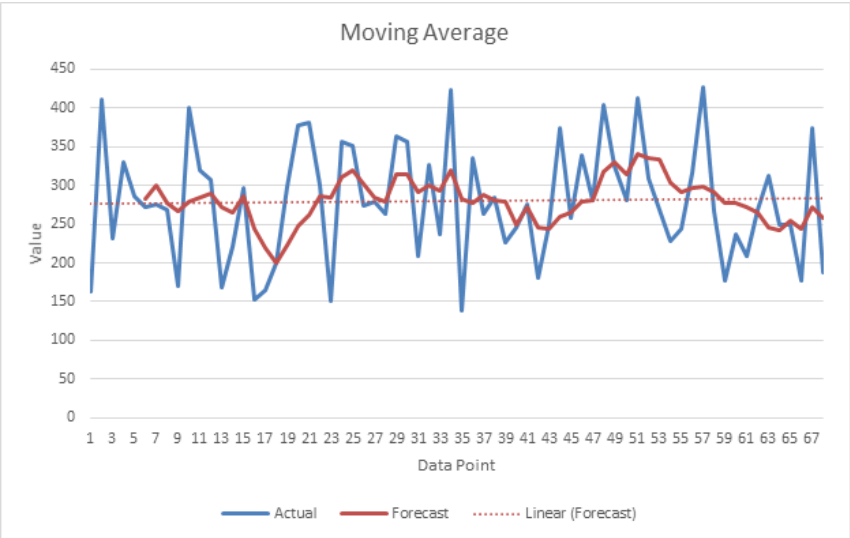


Figura 5.9.: Esempio di filtraggio con media mobile

Termostato digitale

I termostati con la funzione di regolazione della temperatura oraria hanno un'interfaccia simile a quella visualizzata in Figura 5.10.

Figura 5.10.: Esempio di termostato digitale

Come si può vedere, sul fondo del pannello è presente un "grafico" che associa ad ogni ora del giorno (da 0 a 23) una temperatura "obiettivo", cioè una temperatura che il sistema di riscaldamento deve cercare di far raggiungere al locale dove si trova il termostato. Nella figura si può notare, ad esempio, che alle 10 del mattino la regolazione prevede che si provi a raggiungere la temperatura **t2**, mentre alle 21 la temperatura **t1**‡.

Le temperature **t1, t2 e t3** possono essere impostate dall'utente, come si vede nel display in alto a sinistra. Il termostato comanda l'impianto di riscaldamento "dicendogli" se deve accendersi o spegnersi: nella versione semplificata di questo problema si assume che il comando venga dato all'inizio di ogni ora e lo stato dell'impianto di riscaldamento (acceso/spento) rimanga lo stesso per tutta l'ora.

Il funzionamento dell'impianto è il seguente:

- il termostato all'inizio dell'ora accende (o lascia acceso) il riscaldamento se la temperatura attuale dell'ambiente è minore della temperatura obiettivo per quell'ora
- il termostato all'inizio dell'ora spegne (o lascia spento) il riscaldamento se la temperatura attuale dell'ambiente è maggiore o uguale alla temperatura obiettivo per quell'ora
- se il riscaldamento è acceso la temperatura dell'ambiente alla fine dell'ora sarà maggiore di quella di partenza di 0.8 C°
- se il riscaldamento è spento la temperatura dell'ambiente alla fine dell'ora sarà minore di quella di partenza di 0.5 C°

Scrivere una funzione che abbia come parametri il grafico delle temperature obiettivo (sostanzialmente un elenco di 1, 2 o 3 che rappresentano quale temperatura si vuole raggiungere ora per ora), i valori di t1, t2 e t3 e la temperatura iniziale alle ore 0, e che inserisca in un vettore lo stato

‡ Nella figura alcune ore, ad esempio le 6, hanno due temperature obiettivo: ciò è irrilevante ai fini della scrittura della soluzione, in quanto bisogna considerare che ogni ora abbia soltanto una temperatura obiettivo.

dell'impianto di riscaldamento (acceso/spento) ora per ora, seguendo le regole sopra descritte. Un esempio di soluzione, in cui vengono mostrate 6 ore anzichè 24, è il seguente:

```
Temperatura iniziale: 16.5 C°
t1 = 20.8 C°, t2 = 18.0 C°, t3 = 16.5 C°
```

grafico	3	3	1	1	2	3
stato riscaldamento	0	1	1	1	0	0

Le stringhe 6.

Introduzione

Le stringhe sono *oggetti* di uso estremamente comune nella programmazione, anche se finora sono state incontrate solo marginalmente, sotto forma di stringhe costanti racchiuse da virgolette doppie. In informatica con il termine generico *stringa* ci si riferisce a una sequenza di caratteri che sia interpretabile dagli umani come una parola, una frase, un testo più o meno lungo.

Definizione di stringa

Una stringa è una sequenza di caratteri utilizzata per rappresentare una parola, una frase, un qualsiasi testo più o meno lungo.

Esempi di stringhe sono i messaggi inviati su un social network, una pagina web prima che il browser la trasformi in una serie di elementi grafici e anche il codice sorgente di un programma come quelli scritti finora altro non è che una lunga stringa di testo che il compilatore trasforma in un codice eseguibile.

Nel linguaggio C++, a causa della retrocompatibilità con il linguaggio C, le stringhe possono essere gestite utilizzando due approcci diversi:

- come vettori di caratteri *terminati* dal carattere zero binario
- come oggetti della classe di libreria `string`

Il primo approccio è tendenzialmente più complesso e verrà illustrato rapidamente solo per permettere una comprensione basilare in quei contesti (librerie o programmi C) nei quali ne é obbligatorio l'utilizzo. Laddove possibile saranno quindi preferite le `string` del C++, poichè più robuste e in grado di permettere la scrittura di codice più semplice e più leggibile.

6.1. Le stringhe del C

Come detto, le stringhe in C non sono altro che vettori di caratteri e, come tali, è necessario definirne la dimensione in fase di dichiarazione, come per tutti i vettori. A differenza dei vettori numerici, nei quali è necessario sapere sempre la lunghezza della parte «valida»,

cioè quanti sono i primi `n` elementi del vettore che corrispondono effettivamente a dei dati, per quanto riguarda le stringhe, invece, i progettisti del linguaggio C hanno deciso che ogni stringa C valida abbia come *terminatore* un carattere speciale, lo zero binario, che ne indica la fine.

Da questa scelta deriva il modo in cui vengono trattate le stringhe in qualsiasi funzione di libreria e derivano, purtroppo, una serie di possibilità di errore che, nel corso dei decenni passati e ancora attualmente, rendono complessa la scrittura di codice a prova di errore. Basti vedere il listato 6.1 per rendersi conto delle insidie nascoste: dopo la dichiarazione della variabile `stringa` viene eseguito l'input con il solito meccanismo che utilizza la `std::cin` e che permette all'utente di digitare la parola che vuole inserire, confermandola con la pressione del tasto Invio. L'esempio sembra innocuo e, nel caso si inserisca la parola *ciao*, il programma stamperà la frase «*Hai inserito la parola: ciao*». Il problema, in questo esempio, nasce quando si inserisce una parola più lunga di 9 caratteri[1]: infatti i caratteri verranno comunque inseriti nella zona di memoria riservata alla variabile `stringa`, essendo però più di 9 il comportamento del programma sarà indeterminato (e quindi sbagliato), allo stesso modo di quello che succede se si prova a inserire un elemento in un vettore oltre ai limiti definiti dalla sua dimensione.

1: Si ricordi che un carattere deve essere riservato al terminatore.

Listing 6.1: Dichiarazione di una stringa in C e successivo I/O.

```
1   int main() {
2       const int MAX = 10;
3       char stringa[MAX];
4       std::cout << "Inserisci una parola";
5       std::cin >> stringa;
6       std::cout << "Hai inserito la parola: " <<
7           stringa << std::endl;
8       return 0;
9   }
```

Per questo e per molte altre problematiche più complesse, che derivano comunque dal fatto che le stringhe C devono essere trattate a un livello molto basso come vettori, si preferirà utilizzare le `string` introdotte dal C++.

La manipolazione dei caratteri

Prima di passare alle stringhe del C++, si spenderanno due parole su cosa si intenda per *carattere* nei linguaggi C/C++. Senza entrare nel dettaglio, la gestione dei simboli che rappresentano le lettere in un qualsiasi alfabeto è decisamente complessa. Ai tempi dell'invenzione del C, non era ancora così chiaro come sarebbero evoluti i computer e, anche se probabilmente i progettisti dell'epoca si saranno immaginati che la gestione dei caratteri in

modo universale sarebbe stato qualcosa di rognoso e complesso, decisero di semplificare notevolmente il problema, limitandosi a rappresentare solo le lettere minuscole e maiuscole degli alfabeti occidentali, le cifre e i caratteri più comuni (+ , - ; ? ! ecc.). Siccome nei computer si possono rappresentare solo bit, l'idea fu quella di associare dei simboli a dei numeri binari , come si può vedere nella Figura 6.1, definendo uno standard denominato ASCII[2] (American Standard Code for Information Interchange). I progettisti del C quindi decisero di usare un solo byte per rappresentare i caratteri, cosa che è rimasta inalterata e che viene mantenuta anche nelle `string` del C++. Basti a questo punto sapere che finché ci si limita ad utilizzare stringhe che non contengono caratteri esterni al codice ASCII non si incontreranno mai problemi[3], per il resto si lascia all'interesse del lettore l'approfondimento di tematiche quali, solo a titolo di esempio, possono essere lo standard UNICODE e la codifica `utf-8`.

2: Questo standard venne deciso all'inizio degli anni '60.

3: Avendo l'italiano ad esempio le lettere accentate, spesso queste possono generare dei problemi, in dipendenza a molti fattori, che non verranno analizzati in questo testo.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
32	20	[SPACE]	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	[DEL]

Figura 6.1.: Parte della tabella ASCII

Vale la pena far notare che i caratteri maiuscoli e quelli minuscoli sono rappresentati con valori diversi, ad esempio la **A** è il valore 65, mentre la **a** è il valore 97. Solitamente nel codice i valori numerici dei codici ASCII non vengono mai usati, ma è importante sapere che i caratteri nel computer in realtà sono dei numeri interi, poiché ciò permette di effettuare alcune operazioni elementari sui caratteri, come ad esempio la trasformazione di una maiuscola in una minuscola o viceversa, attraverso delle operazioni tra numeri interi, come si può vedere nel listato 6.2, dove la stringa «*albero*» viene trasformata nella stringa «*ALBERO*».

```
1    int main() {
```

Listing 6.2: Conversione di una parola da lettere tutte minuscole a lettere tutte maiuscole.

```
        char stringa[] = "albero";
        for (int i = 0; stringa[i] != '\0'; ++i){
            stringa[i] = stringa[i] - 'a' + 'A';
        }
        std::cout << stringa << std::endl;
        return 0;
    }
```

Come si vedrà la stessa operazione è indifferente al tipo di stringa utilizzata e quindi funzionerà anche con le stringhe del C++.

6.2. Le stringhe del C++

Le `string` del C++ sono delle stringhe con i «superpoteri», poichè le parti che possono creare problemi, come la gestione della memoria e l'accesso ai singoli caratteri, vengono gestite in un modo che elimina alla radice questi problemi, come si vedrà.

Il primo aspetto da notare è che le `string` si comportano come un tipo di dato ordinario, nel senso che la dichiarazione, come si può vedere nel 6.3 è del tutto simile a quella di `int`, un `float` o un qualsiasi altro tipo nativo e la stessa cosa varrà anche per il passaggio di parametri, il valore di ritorno, la dichiarazione di vettori di stringhe, ecc.

Listing 6.3: Dichiarazione di `string`.

```
    //Dichiarazione di una stringa
    std::string s;
    //Dichiarazione e inizializzazione di una stringa
    std::string t = "esempio";
```

Input e output

La gestione dell'input e dell'output in generale può essere fatta come per qualsiasi altro tipo di dato, finchè si tratta di leggere **una sola** parola. Come si può vedere nel listato 6.4, l'input utilizza `std::cin` per la lettura della parola, mentre l'output viene fatto attraverso l'utilizzo di `std::cout`.

Listing 6.4: I/O di `string`.

```
    std::cout << "Inserisci una parola: ";
    std::cin >> s;
    std::cout << "La parola che hai inserito è " <<
        s << std::endl;
```

L'unica difficoltà di questo meccanismo la si incontra nel momento in cui si voglia permettere all'utente di inserire una frase, cioè un insieme di parole separate da spazi. In questo caso l'utilizzo di `cin` causerebbe la lettura della prima parola della frase, mentre tutte quelle seguenti rimarrebbero nel *buffer* di input, causando problemi nelle successive letture. Per verificare questo comportamento è

sufficiente eseguire il listato 6.4: si vedrà che inserendo una sola parola tutto funzionerà correttamente, mentre l'inserimento di una frase causerà il comportamento anomalo conseguenza della problematica appena descritta.

Listing 6.5: Problemi nella lettura di una frase.

```
1     std::cout << "Inserisci una parola: ";
2     std::cin >> s;
3     std::cout << "La parola che hai inserito è " <<
4         s << std::endl;
5     int n;
6     std::cout << "Inserisci un numero: ";
7     std::cin >> n;
8     std::cout << "Il numero che hai inserito è " <<
9         n << std::endl;
```

Siccome non è realistico pensare di non dover mai inserire delle frasi in un programma, la soluzione è quella di utilizzare un altro meccanismo di input, in particolare quello che utilizza la funzione di libreria `getline`, come si può vedere nel listato 6.6. Questa funzione prende come parametri il **`cin`** e la stringa dentro la quale si vuole inserire la frase: la sua chiamata metterà il programma in attesa dell'input dell'utente, che potrà inserire una frase lunga a piacere, al termine della quale premerà il tasto Invio e la frase, fino alla fine della riga, verrà memorizzata nella stringa passata come secondo parametro. Tutto bene? Quasi, nel senso che se si osserva le righe dalla 5 alla 9 ci si accorge che, rispetto all'esempio precedente, adesso viene prima letto il valore del numero da inserire all'interno della stringa temporanea `temp` utilizzando sempre la `getline` e successivamente, alla linea 9, questa stringa viene convertita in un intero utilizzando la funzione `stoi`. Perchè non si poteva utilizzare ancora **`cin`**? Il motivo risiede nel fatto che `getline` e **`cin`** trattano la gestione del fine riga in modo diverso e questo genera quasi sicuramente dei comportamenti indesiderati nel caso si provino a usarle entrambe nello stesso programma. La modalità corretta è quindi quella di utilizzare o solo il **`cin`** o solo la `getline`. Siccome però la `getline` permette solo di leggere stringhe, se lo scopo è quello di leggere dei dati numerici, sarà necessario leggere l'input all'interno di una stringa e successivamente convertirlo nel tipo di dato desiderato utilizzando le funzioni della famiglia `sto*`, dove al posto dell'asterisco andrà la lettera `i` se si vuole convertire la stringa in un **`int`**, come nell'esempio, la `f` se la si vuole convertire in un **`float`** e la `d` se la si vuole convertire in un **`double`**.

Listing 6.6: Utilizzo di getline.

```
1     std::cout << "Inserisci una parola: ";
2     std::getline(std::cin, s);
3     std::cout << "La parola che hai inserito è " <<
4         s << std::endl;
5     int n;
6     std::string temp;
7     std::cout << "Inserisci un numero: ";
```

```
8     std::getline(std::cin, temp);
9     n = std::stoi(temp);
10    std::cout << "Il numero che hai inserito è " <<
11        n << std::endl;
```

Operazioni elementari

Un altro dei motivi per cui la classe `string` risulta comoda da utilizzare risiede nel fatto che è possibile utilizzare molti degli operatori che già vengono utilizzati per i tipi di dati nativi. Degli esempi elementari possono essere visti nel listato 6.7: l'assegnamento viene fatto utilizzando l'operatore = come per tutti gli altri tipi di dati, sia per l'assegnamento di valori costanti, come alle righe 1-2, che per l'assegnamento tra stringhe, come alla riga 8. Anche per le operazioni di confronto possono essere usati gli operatori `==`, `!=`, `<`, `<=`, `>`, `>=`: in questo caso il criterio di ordinamento viene detto *lessicografico* e altro non è che l'ordinamento che si trova nel vocabolario, dove una parola è minore di un'altra se viene prima nel vocabolario. Da notare che, essendo le stringhe composte da caratteri ASCII, le lettere maiuscole e le lettere minuscole sono considerate diverse e una lettera maiuscola viene prima della stessa lettera minuscola, quindi, ad esempio «Albero» viene prima di «albero». Nell'esempio si può vedere l'utilizzo del *minore* alla riga 3. Un'ultima operazione elementare che viene spesso utilizzata dai programmi che trattano stringhe, è l'operazione di *concatenazione*: questa operazione unisce tra loro due stringhe in modo che la stringa risultante sia composta dalla prima stringa seguita immediatamente dalla seconda e viene utilizzato l'operatore + per indicarla. Nel listato 6.7 alla riga 8 si può vedere la concatenazione delle due stringhe `s` e `t` aggiungendo anche la stringa formata dalla lettera *e* con uno spazio prima e uno dopo, che formerà quindi la stringa «arancia e fragola».

Listing 6.7: Operazioni elementari su stringhe.

```
1     std::string s = "arancia";
2     std::string t = "fragola";
3     if (s < t)
4         std::cout << s << " viene prima di " <<
5             t << std::endl;
6     else
7         std::cout << s << " viene dopo " << t << std::endl;
8     std::string v = s + " e " + t;
9     std::cout << v << std::endl;
```

Accesso ai caratteri di una stringa

Spesso è utile accedere ai singoli caratteri che formano una stringa, allo stesso modo in cui è utile accedere ai singoli elementi di un vettore numerico. Con le stringhe stile-C si era già visto in

precedenza nel listato 6.2 che la sintassi era quella dei vettori C e l'accesso si otteneva con le parentesi quadre. Anche le **`string`** permettono l'accesso utilizzando le parentesi quadre, ma viene fornito un altro meccanismo considerevolmente più robusto, che prevede l'utilizzo del *metodo* `at`. Un metodo[4] non è altro che una funzione speciale che viene applicata a un particolare tipo di variabile, in questo caso una **`string`**, e che serve ad effettuare una qualche operazione su questa variabile. Il metodo `at` permette di ottenere lo stesso effetto delle parentesi quadre, come si può vedere alla riga 2 (lettura del valore con indice 2) e alla riga 3 (scrittura di una 'o' all'indice 3), ma aggiunge un controllo sull'indice passato: infatti, se non risultasse valido, perchè negativo o perchè maggiore del dimensione della stringa, porterà alla chiusura del programma con un messaggio d'errore. Nel listato 6.8 si possono vedere alcuni esempi di accesso ai caratteri: in particolare l'esecuzione della riga 6 porterà alla terminazione del programma con il messaggio[5]:

4: I metodi verranno ampiamente approfonditi nel secondo volume, ma essendo il C++ un linguaggio ad oggetti vale comunque la pena di utilizzare alcune delle sue caratteristiche anche se potrebbe non essere completamente chiaro il significato.

```
terminate called after throwing an instance of
'std::out_of_range'
what():  basic_string::at: __n (which is 7)
>= this->size() (which is 7)
```

5: L'esatto messaggio dipende dal compilatore, ma la sostanza è la stessa: in questo caso si è tentato l'accesso al carattere 7 di una stringa di lunghezza 7, che quindi non è valido, in quanto l'indice valido di valore massimo è 6.

In questo listato si può anche vedere una modalità tipica per iterare su tutti i caratteri della stringa: alla riga 7 viene utilizzato un **`for`** il cui indice arriva fino alla dimensione della stringa, ottenuta tramite un altro metodo, il metodo `size()`. Questa modalità garantisce la validità dell'indice e in questo esempio permette di stampare ogni singola lettera della stringa `s`, sia come carattere che nella sua codifica ASCII.

Listing 6.8: Accesso ai caratteri di una stringa.

```
1   std::string s = "arancia";
2   std::cout << "Lettera in posizione 3 è una " << s.at(2)
     << std::endl;
3   s.at(6) = 'o';
4   std::cout << "Adesso l'arancia è diventata un " << s <<
     std::endl;
5   //Questa istruzione causa un'eccezione e fa terminare
    il programma
6   std::cout << s.at(7) << std::endl;
7   for (int i = 0; i < s.size(); ++i) {
8       std::cout << "s.at(" << i << ") -> " << s.at(i) <<
9           " in ASCII " << (int)s.at(i) << std::endl;
10  }
```

6.3. Metodi principali

Con quanto visto finora si potrebbero già risolvere problemi più o meno complessi sulle stringhe, ma lavorare al livello dei singoli

Tabella 6.1.: Principali metodi delle stringhe.

Metodo	Spiegazione
`s.find(const string& cercata)`	Cerca la stringa `cercata` all'interno di `s` e ritorna la posizione della prima occorrenza, `std::string::npos` se non è presente
`s.find(const string& cercata, int pos)`	Cerca la stringa `cercata` all'interno di `s` a partire da `pos` e ritorna la posizione della prima occorrenza, `std::string::npos` se non è presente
`s.substr(int pos)`	Ritorna la sottostringa di `s` a partire dal carattere in posizione `pos` fino alla fine di `s`
`s.substr(int pos, int lun)`	Ritorna la sottostringa di `s` a partire dal carattere in posizione `pos` e di lunghezza pari a `lun`
`s.insert(int pos, const string& nuova)`	Inserisce la stringa `nuova` all'interno di `s`, a partire dalla posizione `pos`
`s.erase(int pos, int lun)`	Cancella la parte di `s` che inizia in posizione `pos`, per `lun` caratteri
`s.replace(int pos, int lun, const string& nuova)`	Sostituisce la parte di `s` che inizia in posizione `pos`, per `lun` caratteri, inserendo al suo posto `nuova`

caratteri è adeguato quando il problema è molto particolare, altrimenti il rischio è di perdersi nei dettagli e rischiare di scrivere codice facilmente soggetto a errori. La libreria delle stringhe del C++ offre invece una serie di *metodi*, di utilizzo generale, che verranno mostrati nei prossimi paragrafi e che permettono di semplificare la scrittura di molti programmi. Una panoramica riassuntiva si trova nella tabella 6.1.

Ricerca in una stringa

Una funzionalità molto generale è quella di cercare l'*occorrenza* di una stringa all'interno di un'altra, basti pensare alla ricerca di una parola in un testo o di una pagina web, funzionalità così comune che moltissimi programmi diversi tra di loro permettono di attivarla con la combinazione ctrl + f. Il metodo che permette di ottenere questo risultato si chiama `find` e può essere utilizzato in due modi leggermente diversi[6]:

- con un solo parametro, che è la stringa da cercare: il metodo poi ritornerà la posizione della prima *occorrenza* nel caso la trovi, oppure `std::string::npos` in caso contrario
- con un ulteriore parametro, che indica da che posizione si intende partire per la ricerca: il valore di ritorno sarà lo stesso della prima versione.

Nel codice 6.9 si può vedere il primo utilizzo alla riga 4, dove la parola «mela» verrà trovata all'interno di `s` in posizione 11, considerando sempre che la prima lettera è in posizione 0. Nel caso invece della ricerca della parola «arancia», che viene cercata

6: In realtà il metodo è uno solo, ma utilizzando i *parametri di default* sembra che in effetti i metodi siano due, uno con un parametro e uno con due.

alla riga 7, il risultato sarà negativo e quindi il metodo ritornerà la posizione `std::string::npos`[7], per indicare che la parola non è presente. Quando usare la versione con anche la posizione di partenza? Un esempio tipico lo si può vedere alle righe 12-17: siccome il metodo `find` trova solo la prima *occorrenza* di una stringa, se ce ne fosse più di una non sarebbe possibile trovare le altre, poichè, anche se chiamato più volte, il metodo ritornerebbe sempre la posizione della prima trovata. Per risolvere questo problema è sufficiente utilizzare un ciclo `while` la cui condizione chiama il metodo `find` a partire da `pos`, con `pos` che inizialmente vale 0, e memorizza la posizione nella stessa variabile. Se questa è diversa da `std::string::npos` allora vuol dire che la stringa è stata trovata, quindi si farà quello che si vuole, in questo esempio semplicemente stampare la posizione, e si aggiornerà il valore di `pos` di 1, in modo che al prossimo ciclo il metodo `find` ripartirà a cercare *dopo* il punto dove è stata trovata la parola: se quindi ce ne fosse un'altra verrà trovata e le operazioni verranno ripetute fino a quando non sarà più possibile trovare una nuova occorrenza.

7: Questa scrittura denota un valore definito dalla libreria che solitamente coincide con -1, ma per motivi di portabilità viene definito in questo modo per poter essere usato su tutte le architetture senza dipendere dalla rappresentazione di uno specifico valore.

Listing 6.9: Utilizzo del metodo `find`.

```
1     int pos;
2     std::string s = "Compra una mela e una pera.";
3     std::string cercata = "mela";
4     pos = s.find(cercata);
5     std::cout << "La parola " << cercata <<
6     " si trova in posizione " << pos << std::endl;
7     std::string non_presente = "arancia";
8     pos = s.find(non_presente);
9     std::cout << "La parola " << non_presente <<
10        " non c'è e la posizione è quindi " <<
11        pos << std::endl;
12    pos = 0;
13    cercata = "una";
14    while ((pos = s.find(cercata, pos)) !=
15            std::string::npos){
16            std::cout << cercata << " trovata in posizione
      " <<
17            pos << std::endl;
18        pos++;
19    }
```

Estrazione di una sottostringa

Un'altro metodo che può tornare utile quando si lavora con le stringhe è il metodo `substr`, che permette di estrarre una parte di una stringa, in uno di questi due modi:

- con un solo parametro, che indica la posizione da cui partire per estrarre la sottostringa, che ritornerà la sottostringa che parte dalla posizione indicata e arriva fino alla fine della stringa a cui è applicato il metodo

- con un ulteriore parametro, che indica per quanti caratteri deve essere estesa la sottostringa da estrarre: in questo caso il valore di ritorno sarà la sottostringa che parte dalla posizione indicata e ha lunghezza pari a quella contenuta nel secondo parametro.

Nel codice 6.10 si può vedere come questa funzione viene usata per estrarre la parte del dominio in un indirizzo email: siccome un indirizzo email è formato da un identificativo dell'utente, da un carattere @ e dal successivo nome di dominio, si può utilizzare il metodo `find` per trovare la posizione della @ e, con `substr`, si può estrarre il nome di dominio partendo da una posizione successiva alla @ (l'istruzione `pos + 1` serve appunto a evitare di inserire la @ nel nome di dominio) e arrivando fino alla fine della stringa. Volendo invece estrarre solo il nome del provider, in questo esempio `gmail`, si può utilizzare `find` nella seconda forma per trovare dove si trova il punto nel nome di dominio (riga 6) e successivamente estrarre solo la sottostringa che va dalla @ al punto. Come si può vedere alla riga 7, il secondo parametro, cioè la lunghezza della stringa da estrarre, viene calcolata sottraendo la posizione della @, indicata con `pos`, dalla posizione del punto, indicata con `punto` e poi sottraendo 1, poichè nè la @ nè il punto devono essere inclusi.

Listing 6.10: Utilizzo del metodo `substr`.

```
1    std::string s = "alessandro.bugatti@gmail.com";
2    int pos = s.find("@");
3    std::string dominio = s.substr(pos + 1);
4    std::cout << "Il dominio della mail è " << dominio <<
5        std::endl;
6    int punto =s.find(".", pos);
7    std::string nome = s.substr(pos + 1, (punto - pos - 1))
     ;
8    std::cout << "Il nome senza il .com è " << nome <<
9        std::endl;
```

Modifiche a una stringa

Uno degli insiemi più comuni di operazioni che vengono effettuate sulle stringhe sono quelle che modificano una stringa per inserire nuove parole, cancellarne delle parti o sostituirne altre.

Per evidenziare quanto spesso si incontrano in pratica, basti pensare a una delle attività che dovrebbe praticare chi sta leggendo questo testo: scrivere codice. Quando si scrive codice è molto frequente tagliare un pezzo di codice (`erase`), per incollarlo successivamente in una nuova posizione (`insert`), oppure chiedere all'editor di cambiare il nome di una variabile in tutti i punti dove viene utilizzata (`replace`).

Listing 6.11: Utilizzo dei metodi `erase`, `insert` e `replace`.

```
1    std::string frase = "I computer sono inutili.";
```

```
    frase.erase(16, 2);
    std::cout << frase << std::endl;
    frase.insert(21, ", a volte");
    std::cout << frase << std::endl;
    frase.replace(2, 8, "programmi");
    std::cout << frase << std::endl;
```

Nel listato 6.11 si può vedere un semplice esempio che utilizza tutti e tre i metodi indicati. Come si vede, la cancellazione di una parte della stringa prevede come parametri il punto da cui partire e in numero di caratteri, da cancellare, in questo esempio 2, e quindi la stringa verrà trasformata in «*I computer sono utili.*».

L'inserimento prevede invece come parametri il punto dove inserire la nuova parola/frase e la parola/frase da inserire, che può essere constante, come in questo esempio, oppure una variabile, sempre di tipo `string`. Il metodo si occupa di "creare spazio" per fare in modo che la stringa inserita stia all'interno della stringa modificata, che quindi viene allargata in modo da contenerla. Nell'esempio, la stringa «*I computer sono utili.*» diventerà «*I computer sono utili, a volte.*».

Infine volendo sostituire una parte di una stringa con un'altra, il metodo `replace` prevede di indicare il punto di inserimento e il numero di caratteri da sostituire, in questo esempio 8, e poi la stringa da sostituire, dove in questo esempio «computer» verrà sostituito dalla parola «programmi», creando la frase «*I programmi sono utili, a volte.*». La differenza con il metodo `insert` è che, mentre `insert` inserisce una parola all'interno della stringa, lasciando tutto ciò che era già presente, il metodo `replace` elimina anche parte della stringa originale, sostituendola con qualcos'altro. Si potrebbe notare che l'effetto della chiamata a `replace` si potrebbe anche ottenere con una combinazione dei metodi `erase` e `insert`, ma l'utilizzo di `replace` risulta più intuitivo e più comodo.

Da notare che nessuna di queste operazioni, se si provasse a reimplementarla lavorando sui singoli caratteri della stringa, sarebbe banale, quindi il consiglio è di lavorare sui singoli caratteri laddove ci sia un'esigenza molto specifica e utilizzare invece questi metodi nei casi generali.

6.4. Stringhe e funzioni

Per quanto riguarda il passaggio di stringhe come parametri a una funzione e la possibilità di ritornare una stringa come valore di ritorno da una funzione, nulla cambia rispetto a quanto visto con i tipi primitivi (`int`, `float`, ecc.). Per questioni di ottimizzazione, che però spesso si trovano nelle funzioni di libreria e che quindi

è bene conoscere, il passaggio di una variabile `string` avviene in generale per *riferimento* e spesso indicando che non è possibile modificarla, utilizzando la parola chiave `const`. Il motivo del passaggio per riferimento risiede nel fatto che la copia di una stringa è un'operazione relativamente onerosa, quindi si preferisce evitarla. Siccome però il passaggio per riferimento potrebbe portare alla generazione di errori difficili da scoprire perchè non individuabili dal compilatore, l'aggiunta dell'indicazione `const`, risolve questo problema indicando al compilatore che l'intenzione è quella di non modificare la stringa a cui viene applicato, in modo da permettere controlli in fase di compilazione[8]. Un esempio di funzione che ha come parametri delle stringhe e ritorna una stringa si può vedere nel codice 6.12: scopo della funzione è quello di prendere due stringhe, passate come riferimenti costanti, e ritornare una stringa ottenuta come concatenazione delle due per formare un indirizzo email.

8: I controlli in fase di compilazione rendono il codice più robusto, evitando, dove possibile, i più insidiosi errori in *runtime*.

Listing 6.12: Funzione che ha due stringhe come parametri e ritorna una stringa.

```
1    std::string crea_email(const std::string& utente,
2                           const std::string& dominio) {
3        return utente + '@' + dominio;
4    }
```

Per vedere gli effetti della mancanza della parola `const` si può vedere il codice 6.13: anche se all'apparenza il codice è del tutto simile, a parte la mancanza della parola `const`, l'itruzione che ritorna la stringa contiene un problema non semplice da vedere. Infatti, in questo caso, è stato usato `+=` al posto del `+` del listato 6.12: un effetto collaterale non evidente è che verrà modificato il contenuto della variabile `utente`, che essendo passata come riferimento, verrà modificata nel contesto dove la funzione è stata chiamata. Un errore del genere potrebbe causare problemi non semplici da individuare, poichè comunque la funzione ritornerebbe il valore corretto e la modifica potrebbe passare inosservata e palesarsi solo in certi rami di esecuzione. Utilizzando `const` invece il problema verrebbe rilevato in tempo di compilazione, impedendo la generazione dell'eseguibile e obbligando il programmatore a interrogarsi su quanto scritto.

Listing 6.13: Problema con i riferimenti.

```
1    std::string crea_email(std::string& utente,
2        std::string& dominio) {
3        return utente += '@' + dominio;
4    }
```

6.5. Stringhe e vettori

Come per le funzioni, anche quando si parla di vettori non ci sono differenze rispetto ai vettori già visti per i tipi primitivi, sia

in termini di dichiarazione, di utilizzo attraverso gli indici e di passaggio di parametri a funzione.

Listing 6.14: Utilizzo di vettori di stringhe.

```
int quante_frasi(const std::string frasi[], int n,
    const std::string& parola){
    int contatore = 0;
    for (int i = 0; i < n; ++i) {
        if (frasi[i].find(parola) != std::string::npos)
            contatore++;
    }
    return contatore;
}

int main() {
    const int N = 3;
    int contatore;
    std::string frasi[N], parola;
    std::cout << "Inserisci " << N << " frasi" <<
        std::endl;
    for (int i = 0; i < N; ++i) {
        std::getline(std::cin, frasi[i]);
    }
    std::cout << "Inserisci una parola: ";
    std::getline(std::cin, parola);
    contatore = quante_frasi(frasi, N, parola);
    std::cout << "La parola " << parola <<
        " è contenuta in " << contatore <<
        " frasi su " << N << std::endl;
    return 0;
}
```

Lo stesso si può dire per i `vector`, come si può vedere dal seguente listato, che non è altro che il listato 6.14, dove al posto dei vettori nativi vengono usati i `vector`

Listing 6.15: Utilizzo di `vector` di stringhe.

```
int quante_frasi(const std::vector<std::string>& frasi,
const std::string& parola){
    int contatore = 0;
    for (int i = 0; i < frasi.size(); ++i) {
        if (frasi.at(i).find(parola) != std::string::
npos)
            contatore++;
    }
    return contatore;
}

int main() {
    const int N = 3;
    int contatore;
    std::vector<std::string> frasi(N);
    std::string parola;
    std::cout << "Inserisci " << N << " frasi" <<
    std::endl;
```

```
18          for (int i = 0; i < N; ++i) {
19              std::getline(std::cin, frasi.at(i));
20          }
21          std::cout << "Inserisci una parola: ";
22          std::getline(std::cin, parola);
23          contatore = quante_frasi(frasi, parola);
24          std::cout << "La parola " << parola <<
25          " è contenuta in " << contatore <<
26          " frasi su " << N << std::endl;
27          return 0;
28      }
```

6.6. Esercizi

Studi

1. Scrivere un programma che faccia inserire un nome e un cognome e poi stampi la frase "Buongiorno nome cognome", dove al posto di nome e cognome vanno stampati il nome e il cognome inseriti.
2. Scrivere un programma che faccia inserire due parole e poi stampi quella più lunga.
3. Scrivere un programma che faccia inserire due parole e poi stampi quella che viene prima in ordine alfabetico.
4. Scrivere un programma che faccia inserire una stringa e la ristampi con tutte le lettere separate dal segno **-**.
5. Scrivere una funzione che prenda una stringa come parametro e ritorni il numero di vocali contenute al suo interno. Inizialmente si supponga che la stringa contenga solo caratteri minuscoli, successivamente si riscriva la funzione togliendo questa ipotesi. Si scriva poi un programma che permetta di testarne il funzionamento.
6. Scrivere una funzione che prenda una stringa come parametro e ritorni il numero di consonanti contenute al suo interno, supponendo che contenga solo vocali e consonanti e nient'altro. Si scriva poi un programma che permetta di testarne il funzionamento.
7. Scrivere una funzione che prenda una stringa come parametro e ritorni il numero di parole contenute al suo interno, supponendo che ogni parola sia separata dalla seguente da uno spazio. Si scriva poi un programma che permetta di testarne il funzionamento.
8. Si scriva una funzione che, dati come parametri un carattere **c** e un numero intero **n**, restituisca una stringa composta dal carattere **c** ripetuto **n** volte. Si scriva poi un programma che permetta di testarne il funzionamento.
9. Si scriva una funzione che, dati come parametri due stringhe, restituisca la stringa ottenuta alternando tutti i caratteri della prima stringa con quelli della seconda. Nel caso una delle due fosse più corta dell'altra il risultato conterrà alla fine le lettere avanzate dalla stringa più lunga. Esempio: se le due stringhe fossero "*ammirevole*" e "**liuto**" la stringa risultato sarebbe "*a***l***m***i***m***u***i***t***r***o***evole*". Si scriva poi un programma che permetta di testarne il funzionamento.
10. Si scriva una funzione che, dati come parametri una stringa **s** e un numero intero **n**, restituisca una stringa contenente le prime **n** lettere di **s**. Si scriva poi un programma che permetta di testarne il funzionamento.
11. Scrivere un programma che faccia inserire una stringa e estragga la sottostringa che parte dalla metà della stringa immessa fino alla sua fine usando il metodo `substr` e poi la stampi. Esempio: se la parola fosse «*varicella*» il programma dovrà stampare «*cella*».
12. Scrivere un programma che faccia inserire una stringa e estragga la sottostringa ottenuta togliendo il primo e l'ultimo carattere, usando il metodo `substr`, e poi la stampi. Esempio: se la parola fosse «*premio*» il programma dovrà stampare «*remi*».
13. Scrivere un programma che, dato un testo inserito dall'utente, inserisca uno spazio dopo ogni punto che si trova nella frase, tranne eventualmente l'ultimo. Esempio: se la frase fosse «*Sono andato a casa.Non c'era nessuno.Bel problema.Ero senza chiavi.*», diventerebbe «*Sono andato a casa. Non c'era nessuno. Bel problema. Ero senza chiavi.*» (usare `find` e `insert`).
14. Scrivere un programma che, data una frase di lunghezza qualsiasi, sostituisca tutte le occorrenze di una parola *cercata* inserita dall'utente con una parola *nuova*, anch'essa inserita dall'utente (usare `replace` e `find`).

15. Scrivere un programma che, data una frase di lunghezza qualsiasi, elimini tutte le occorrenze di una parola inserita dall'utente (usare `erase` e `find`).

Esercizi

1. Scrivere un programma che, data in input una stringa, ne estragga le vocali e la stampi solo con le consonanti rimaste.
2. Scrivere un programma che trasformi una parola nel codice numerico corrispondente, ottenuto sommando i valori ASCII delle lettere che la compongono. Successivamente l'utente dovrà inserire delle parole fino a quando non ne troverà una diversa con lo stesso codice numerico.
3. Scrivere un programma che generi una password casuale, composta da lettere minuscole, maiuscole e cifre, di una lunghezza decisa dall'utente (usare **+=** per le stringhe e la funzione **rand**)
4. Si scriva una funzione che accetta come parametri una stringa `s` e che restituisce una stringa contenente la parola più lunga che si trova in `s`. Per parola si intende una qualsiasi sequenza di caratteri consecutivi che non contiene spazi. Nel caso il testo contenesse più parole con lo stesso numero massimo di caratteri, verrà presa in considerazione la prima. Si scriva poi un programma per verificare la correttezza della funzione.
5. Si scriva una funzione che accetti come parametro una stringa e che la modifichi in modo che tutti i caratteri non alfabetici vengano eliminati e che eventuali caratteri alfabetici maiuscoli diventino minuscoli. Se ad esempio la stringa fosse "Sono andato a Milano, pioveva.", la stringa diventerà "sonoandatoamilanopioveva". Si scriva successivamente un programma per testare la correttezza della funzione creata.
6. Si scriva una funzione che accetta come parametri una stringa e un intero (chiave di cifratura) e che applichi alla stringa il sistema di cifratura di Cesare. Il cifrario di Cesare sposta avanti ogni lettera del messaggio da cifrare di un numero uguale di posizioni nell'alfabeto in base al valore della chiave: se ad esempio il testo da cifrare fosse «arriviamo» e lo spostamento 3, il testo cifrato diventerebbe «duulyldpr». Si può supporre che le lettere del testo da cifrare siano solo le lettere minuscole dell'alfabeto inglese e quindi il testo non contenga altri tipi di caratteri. La chiave 0 lascia il testo inalterato. Si scriva successivamente un programma per testare la correttezza della funzione creata. Parte aggiuntiva: si scriva poi anche la funzione per decodificare un testo cifrato creato con questo sistema.
7. Si scriva una funzione che accetta come parametri una stringa che contiene un testo da cifrare e una seconda stringa che contiene una chiave di cifratura. Il metodo da utilizzare è la cifratura per sostituzione, nella quale ogni carattere del testo in chiaro viene sostituito da un carattere diverso secondo una regola dettata dalla chiave di cifratura: ad esempio la 'a' diventa una 'r', la 'b' diventa una 'g', la 'c' diventa una 'm', ecc. Un modo compatto di rappresentare la chiave è quello di avere una stringa contenente tutti i 26 caratteri dell'alfabeto inglese in posizioni mescolate, in modo che la lettera che si trova in prima posizione rappresenti il carattere in cui viene trasformata la 'a', quella in seconda posizione il carattere in cui viene trasformata la 'b', ecc. Si può supporre che le lettere del testo da cifrare siano solo le lettere minuscole dell'alfabeto inglese e quindi il testo non contenga altri tipi di caratteri. Si scriva successivamente un programma per testare la correttezza della funzione creata. Parte aggiuntiva: si scriva poi anche la funzione per decodificare un testo cifrato creato con questo sistema.
8. Si scriva una funzione che accetta come parametro una stringa e ritorna una stringa che contenga una sola *occorrenza* di ogni singolo carattere trovato nella stringa passata come

parametro. Se ad esempio la prima stringa contenesse «*Bello questo corso di Informatica*», la stringa ritornata dovrà contenere «*Belo qustcrdiInfmac*». Si scriva successivamente un programma per testare la correttezza della funzione creata.

9. Si scriva una funzione che accetta come parametri una stringa e che ritorna `true` se la stringa contiene la rappresentazione corretta di un numero esadecimale e `false` altrimenti. Un numero esadecimale è corretto se inizia con la stringa `0x` o `0X`, seguita da 1 fino a 8 caratteri che possono essere le 10 cifre decimali e i caratteri dalla A alla F sia maiuscoli che minuscoli. Si scriva successivamente un programma per testare la correttezza della funzione creata.
10. Con la parola «trim» applicata a una stringa si intende solitamente l'operazione di eliminare gli spazi in testa e in coda a una stringa. Si scriva una funzione che accetta come parametro una stringa e la modifica in modo che vengano eliminati eventuali caratteri all'inizio e alla fine della stringa. Per rendere l'esercizio più interessante si può implementare questa funzione sia utilizzando una stringa di appoggio che senza.
    Si scriva successivamente un programma per testare la correttezza delle due funzioni create, quella con la stringa d'appoggio e quella senza.
11. Si scriva una funzione che accetta come parametri un vettore di stringhe, il numero di elementi di cui è composto questo vettore e una stringa `s`. La funzione deve ritornare il numero totale di volte che la stringa `s` compare all'interno del vettore di stringhe. Se ad esempio il vettore di stringhe contenesse le 3 stringhe:

    ```
    la mia macchina è verde e va veloce

    ieri ho pescato una verdesca

    verde come il latte, verde come il sangue
    ```

    e la stringa `s` fosse «verde», la funzione dovrebbe ritornare 4.
    Si scriva successivamente un programma per testare la correttezza della funzione creata.
12. Si scriva una funzione che calcoli la *distanza di Hamming* tra due stringhe. La distanza di Hamming tra due stringhe si ottiene guardando i caratteri delle due stringhe nella stessa posizione e contando quanti di questi sono differenti. Se le due stringhe sono uguali la distanza di Hamming è zero, se ad esempio fossero "Ciao" e "Tiro" la distanza sarebbe 2, difatti sono diverse in posizione 0 e in posizione 2. Se le due stringhe avessero lunghezza diversa la funzione dovrà ritornare -1, poiché il quel caso la distanza di Hamming non è definita. Si scriva successivamente un programma per testare la correttezza della funzione creata.
13. Scrivere un programma che faccia inserire all'utente una stringa di testo, su un'unica riga, di lunghezza qualsiasi e che, successivamente, la ristampi a video dandole una formattazione a "bandiera" con una lunghezza massima di ogni singola riga di 60 caratteri. Formattare *a bandiera* significa che la stringa di partenza andrà stampata in modo che vada a capo in corrispondenza di uno spazio e che ogni singola riga stampata non sia più lunga di 60 caratteri, ma che abbia una lunghezza che sia la massima possibile rispettando i due vincoli precedenti. Per fare un esempio, il primo paragrafo di questo testo risulterebbe formattato in questo modo:

    ```
    Scrivere un programma che faccia inserire all'utente una
    stringa di testo, su un'unica riga, di lunghezza qualsiasi e
    che, successivamente, la ristampi a video dandole una
    formattazione a "bandiera" con una lunghezza massima di ogni
    singola riga di 60 caratteri.
    ```

14. Una semplice codifica lossless per le immagini, chiamata Run-length encoding (RLE), prevede che se l'immagine contiene n caratteri uguali consecutivi, con n > 1, questi vengano sostituiti con il valore n seguito dal singolo carattere che si ripete. Ad esempio

    ```
    dddd -> 4d
    affggg -> a2f3g
    fffffffffffff -> 9f5f
    ```

    Come si vede, nel caso il numero di occorrenze consecutive dello stesso carattere sia maggiore di 9, la sequenza viene spezzata in più parti, in modo da semplificare la fase di decodifica. Sempre per evitare problemi in fase di decodifica si supponga che la stringa da codificare contenga soltanto caratteri, non cifre.
    Si scriva una funzione che accetta come parametro la stringa da codificare e che ritorni una stringa con il testo codificato con l'algoritmo esposto sopra.
    Si scriva successivamente un programma per testare la correttezza della funzione creata.
    Dopo aver verificato la correttezza di quanto fatto, si provi anche a scrivere la funzione che fa la decodifica e la si testi.
15. Si scriva una funzione che accetta come parametro una stringa e restituisce `true` o `false` a seconda che la stringa sia un indirizzo email valido. Ai fini di questo esercizio un indirizzo email é valido se:
    - contiene un solo segno @
    - la parte prima del segno @ deve contenere almeno un carattere e il primo carattere deve essere una lettera
    - la parte dopo il segno @ deve contenere un solo carattere . (punto), che non può essere nè il primo nè l'ultimo carattere.
    - la parte dopo il punto deve essere formata da due o tre caratteri alfabetici.

    Si scriva successivamente un programma per testare la correttezza della funzione creata.
16. Scrivere un programma che chieda all'utente di inserire una stringa di una qualsiasi lunghezza, contenente anche spazi, e stampi il numero di caratteri che compongono ogni singola parola della stringa (si supponga che gli spazi siano i separatori di parole). Ad esempio con la stringa "che esercizio semplice ma non troppo" il programma stamperà 3 9 8 2 3 6.
17. Alcune convenzioni di programmazione prevedono di scrivere i nomi di variabili e funzioni usando la notazione Snake Case (es. *variabile_in_notazione_snake_case*), mentre altre utilizzano la notazione Camel Case (*VariabileInNotazioneCamelCase*). Scrivere una funzione che prenda come parametro una stringa che rappresenta un nome di variabile in notazione Snake Case e la modifichi in modo che rappresenti lo stesso nome usando la convenzione Camel Case. Se ad esempio la stringa passata fosse «conto_corrente_attuale» dovrà essere modificata in «ContoCorrenteAttuale». Si scriva successivamente un programma che chieda all'utente di inserire una stringa contenente un nome in Snake Case (si supponga che il nome inserito sia nella forma corretta) e, utilizzando la funzione precedentemente definita, la modifichi in Camel Case e la stampi a video.
    Si riscriva successivamente un secondo programma che faccia esattamente l'opposto, cioè converta una stringa da Camel Case a Snake Case.
18. In alcuni sistemi per la ricerca di parole all'interno di stringhe di testo, vengono usati dei caratteri *jolly* per permettere una ricerca di parole simili secondo qualche criterio. Si supponga che il carattere '*' sia un jolly che indica che al suo posto può essere messo un *solo* carattere qualsiasi: in questo modo, ad esempio la ricerca della stringa "*co*a*" darebbe esito positivo sia per la stringa *coda* che per *cosa*, ecc.

Scrivere una funzione che, dati come parametri una stringa di testo S, che si può supporre contenere solo caratteri minuscoli o spazi, e un'altra stringa M che rappresenta una *maschera*, cioè una stringa contenente 1 o più asterischi, ritorni il numero di occorrenze all'interno della stringa S che sono compatibili con la maschera M. Ad esempio con S = «quella cosa ha una coda e beve coca-cola» e M = «co*a» la funzione ritornerebbe 4. Scrivere successivamente un programma che faccia inserire due stringhe S e M e, utilizzando la funzione appena definita, stampi a video quante occorrenze di parole contenute in S sono compatibili con M.

19. Il DNA contiene le informazioni per la "costruzione" degli organismi viventi in forma di un codice i cui simboli sono indicati con le quattro lettere A, C, G e T *, indifferentemente maiuscoli o minuscoli. Si scriva una funzione che abbia come parametri una stringa `S` che rappresenta un frammento di DNA e una seconda stringa `T`, rappresentante sempre un frammento di DNA. Questa funzione deve ritornare il numero di volte che la stringa `T` compare all'interno di `S`, senza sovrapposizioni. Ad esempio se `S` fosse "AAGCGTTAGCA" e `T` fosse "AGC" la funzione dovrebbe ritornare il valore 2. Se `S` oppure `T` dovessero contenere un simbolo diverso dai quattro validi, la funzione dovrà tornare -1 per indicare un errore nei frammenti di DNA. Successivamente si scriva un programma per testare la correttezza della funzione realizzata.
20. Sempre a proposito di DNA, si scriva una funzione che abbia come parametri una stringa `V` che rappresenta un frammento di DNA e una seconda stringa che dovrà essere «riempita» con il frammento di DNA complementare a `V`. Un frammento di DNA `S` è complementare a un frammento `V` se ogni base di `S` in posizione i-esima è complementare alla base di `V` in posizione i-esima. L'adenina (A) è complementare alla timina (T) e viceversa, la guanina (G) è complementare alla citosina (C) e viceversa. Se **V** dovesse contenere un simbolo diverso dai quattro validi, la funzione dovrà tornare -1 per indicare un errore nel frammento di DNA e l'impossibilità di creare il frammento complementare di `V`, altrimenti ritornerà 1.
21. Scrivere un programma che faccia inserire all'utente una frase di lunghezza qualsiasi, composta solo da parole e spazi. Successivamente il programma dovrà stampare se la frase è *parolindroma*†. Una frase è parolindroma se, dopo aver contato il numero di lettere di cui è composta ogni parola e scrivendo questi numeri uno di seguito all'altro, il numero che ne risulta è palindromo (cioè letto da destra verso sinistra o da sinistra verso destra è uguale). Si supponga che tutte le parole che compongono la frase abbiano lunghezza minore di 10 caratteri, che il separatore di parole sia unicamente lo spazio e che possa comparire solo internamente alla frase.
22. Tra le misure per il contenimento della pandemia da Coronavirus viene stabilito che ogni cittadino possa recarsi a fare la spesa solo in un preciso giorno della settimana. Per associare ogni cittadino a un giorno specifico si procede in questo modo: si prende il codice della carta d'identità, che è sempre composto da due lettere, cinque cifre e altre due lettere, si estrae la parte centrale, quella formata da cinque cifre e si vede quanto vale il resto della divisione per 7, associando al resto 0 la domenica, al resto 1 il lunedì ecc. Se ad esempio il codice fosse AB12345GH, la parte centrale sarebbe 12345 e il resto della divisione per 7 sarebbe 4, quindi il possessore di quella carta di identità potrebbe fare la spesa solo di giovedì. Scrivere un programma che faccia inserire un codice di carta d'identità, che si può supporre sempre corretto, lo passi a una funzione che ne estragga il resto secondo le istruzioni indicate e infine mostri a video il giorno della settimana in cui si può uscire.
23. Si scriva un programma che data in input una stringa di lunghezza qualsiasi, la riorganizzi in modo che tutte le vocali siano nella prima parte e tutte le consonanti siano nella seconda,

* Adenina, citosina, guanina e timina, rispettivamente

† Nonostante la mia avversità per i neologismi, temo di essermi inventato questa parola, quindi non usatela nelle conversazioni tra amici.

mantenendo l'ordinamento relativo tra le vocali e le consonanti. Se ad esempio la stringa in input fosse *"ciao a tutti"*, diventerebbe *"iaoauicttt"*. La stringa così riorganizzata dovrà poi essere stampata dal programma.
Si può supporre che la stringa di input contenga solo vocali e consonanti minuscole e spazi, senza bisogno di controllare.

24. All'interno di un microchip sono contenuti dei dati sotto forma di stringa, nella forma di sequenze di coppie *attributo:valore* separate da uno spazio, come ad esempio nella stringa *"adn:1971 alt:178 val:10"*, dove *adn* rappresenta l'anno di nascita, *alt* rappresenta l'altezza e *val* rappresenta gli anni di validità del passaporto. Scrivere una funzione che, ricevuto come parametro una stringa della forma indicata, stabilisca se è valida o meno secondo queste regole:

    - devono essere necessariamente presenti gli attributi *adn*, *alt*
    - deve essere presente almeno uno tra *val*, *pes* e *cod*

    Se la stringa è valida la funzione deve ritornare 1, altrimenti 0. Gli attributi nella stringa possono comparire in qualsiasi ordine.
    Successivamente si scriva un programma per testare la correttezza della funzione realizzata.
25. Scrivere una funzione che, data come parametro una stringa, la **modifichi** eliminando tutte le eventuali vocali consecutive che si trovano all'inizio e alla fine della parola. Se ad esempio la stringa fosse **inizio**, dopo l'applicazione della funzione diventerebbe **niz**, se fosse **programma** diventerebbe **programm**, se fosse **aia** non rimarrebbe nessuna lettera.
    Successivamente si scriva un programma che faccia inserire una stringa all'utente, che si può supporre essere formata solo da parole e spazi, e applichi la funzione definita in precedenza ad ogni parola, stampando a video il risultato dell'elaborazione. Ad esempio la frase «*il cane abbaia e il gatto dorme nella nostra aia*» diventerebbe «*l can bb l gatt dorm nell nostr*».
26. Scrivere una funzione che, dati come parametri una stringa S e un intero N, che si può supporre ≥ 0, modifichi la stringa S in modo che sia formata dai suoi primi N caratteri in ordine invertito, seguiti dai rimanenti caratteri sempre in ordine invertito. Se ad esempio S valesse «attraversamento» e N fosse 7, la stringa S diventerà «evarttaotnemasr». Se N fosse uguale o maggiore della lunghezza della stringa, la funzione modificherà la stringa iniziale invertendone tutti i caratteri.
    Successivamente si scriva un programma per testare la correttezza della funzione realizzata.
27. Scrivere un programma che chieda all'utente di inserire, mediante una singola operazione di lettura, una stringa di massimo 9 caratteri composta solo da cifre e che controlli che questa condizione sia soddisfatta e, nel caso non lo sia, faccia reinserire la stringa finché non viene soddisfatta.
    Successivamente il programma dovrà eliminare **una e una sola** cifra in modo che il numero diventi divisibile per 3, scrivendo la posizione della cifra eliminata e il numero ottenuto (se ci fossero più modi per ottenere questo risultato, basta che il programma ne trovi uno qualsiasi). Se non fosse possibile, togliendo una sola cifra, rendere il numero divisibile per 3, il programma dovrà scrivere «Non possibile».
    Se ad esempio la stringa inserita fosse 2345678, basterebbe togliere il 2 ottenendo 345678 che è divisibile per 3, stampando che si è tolta la cifra in posizione 0 ottenendo così 345678 (anche togliendo il 5 si ottiene 234678 che è divisibile per 3, ma basta una sola soluzione). Se invece il numero fosse 777 sarebbe impossibile ottenere un numero divisibile per 3 togliendo una sola cifra.

Progetti

Generatore di password

Partendo dal codice scritto per il generatore casuale di password, scrivere un programma che ne estenda le funzionalità, in particolare:

- faccia scegliere all'utente il tipo di caratteri che si troveranno nella password, dividendoli in *famiglie* e permettendo di scegliere quali famiglie considerare. Le famiglie potrebbero essere:
  - lettere minuscole
  - lettere maiuscole
  - cifre decimali
  - cifre esadecimali
  - caratteri speciali presenti in qualsiasi tastiera
  - un insieme specifico di caratteri inseriti dall'utente

  Ovviamente alcune di queste famiglie sono incompatibili fra loro, ad esempio le cifre decimali e quelle esadecimali.
- verifichi che la password generata contenga almeno un carattere per ogni famiglia scelta dall'utente, in modo da assicurare che questa condizione venga soddisfatta nella password creata.
- se la password generata non soddisfa i gusti dell'utente o se comunque se ne vuole generare un'altra, il programma di default deve permettere di ripetere la generazione con le impostazioni scelte in precedenza, se l'utente invece decidesse di cambiarle deve permettergli di farlo.

Cifrario di Vigenère

Il cifrario di Vigenère non è altro che la combinazione di una serie di cifrari di Cesare, alternati secondo le indicazioni della parola cifrante. Se ad esempio la parola fosse SOLE, il primo carattere del testo in chiaro verrebbe sostituito con il carattere del cifrario di Cesare con chiave 18 (poichè S è la 19^a lettera dell'alfabeto inglese), il secondo con il carattere del cifrario di Cesare con chiave 14 (poichè O è la 15^a lettera dell'alfabeto inglese) e così via, con il meccanismo che si ripete ogni volta che "finisce" la parola cifrante: in questo esempio la quinta lettera del testo in chiaro verrà nuovamente codificata con il cifrario di Cesare associato alla lettera S e così via. Se ad esempio la frase da cifrare fosse «*prontiallattacco*» e la chiave cifrante fosse la parola «*sole*», la frase cifrata diventerebbe «*hfzrlwlpdoexsqns*».

Scrivere un programma che, dopo aver fatto immettere all'utente la frase da cifrare e la chiave di cifratura, produca il testo cifrato.

Possibili migliorie della versione base:

- l'utente può inserire una frase contenente qualsiasi simbolo, devono essere eliminati tutte i simboli che non sono caratteri alfabetici e quelli maiuscoli devono essere trasformati in minuscoli, tutto questo prima della codifica
- deve poter essere fatta la decodifica a partire da un testo cifrato e dalla chiave di codifica
- il programma deve permettere di scegliere di fare la codifica oppure la decodifica, ripetendo il procedimento finché l'utente non decide di chiudere il programma

- il programma deve offrire la possibilità di generare una chiave casuale, di cui l'utente può scegliere la lunghezza, e poi usare quella chiave per codificare il testo inserito dall'utente.

Strutture in C++ 7.

Introduzione

I programmi visti finora permettono principalmente di lavorare su informazioni della realtà che sono ben rappresentabili attraverso numeri, sia interi che reali, oppure con stringhe di testo. Al crescere della complessità dei problemi che si vogliono risolvere e volendo gestire realtà più complesse, dover utilizzare solo i tipi primitivi del linguaggio risulta limitante per il programmatore, in quanto i dati presenti nel codice smettono di avere un diretto parallelismo con quelli presenti nella realtà.

Si supponga, ad esempio, di voler scrivere un programma che si occupi di gestire i voti degli studenti di una scuola. In prima approssimazione si può pensare che le informazioni necessarie siano i dati anagrafici degli studenti, le classi in cui si trovano e i voti che hanno preso. Ognuna di queste informazioni può essere a sua volta scomposta in parti più semplici, fino ad arrivare a dei tipi primitivi, solo che a questo punto ci si ritroverebbe con una miriade di variabili, tutte logicamente collegate tra di loro, ma assolutamente indipendenti dal punto di vista del codice, come si può vedere nel listato 7.1:

Listing 7.1: Esempio di rappresentazione di informazioni complesse.

```
1    std::string nome, cognome;
2    int giorno_nascita, mese_nascita, anno_nascita;
3    int classe;
4    char sezione;
5    std::vector <float> voti;
6    std::vector <std::string> materia;
7    std::vector <int> giorno_voto, mese_voto, anno_voto;
```

Come si può vedere, sono presenti 12 diverse variabili, tutte relative ai voti di un singolo studente e quindi ci si può immaginare che anche la gestione di operazioni semplici come l'I/O o il calcolo della media dei voti del primo quadrimestre, risultino complesse per l'elevato numero di variabili presenti.

Anche la scomposizione del codice in una serie di funzioni risulterebbe problematica, perché probabilmente ogni funzione avrebbe necessità di avere molti parametri e quindi sarebbe complessa da usare. Se poi, come sembra ragionevole e necessario, il programma dovesse gestire una serie di studenti, ognuna di quelle variabile dovrebbe diventare un vettore, rendendo il tutto ancora più macchinoso da gestire.

Per rendere la rappresentazione delle informazioni di interesse all'interno del codice più «vicine» alla percezione del programmatore, è stato quindi introdotto il concetto di *struttura*, che permetterà di definire nuovi tipi di dati, per poter gestire in modo più naturale tutte le informazioni del problema che si vuole affrontare.

7.1. Le strutture

Scopo delle strutture è quello di definire dei nuovi tipi del linguaggio, che, appoggiandosi sui tipi nativi e componendoli in maniera opportuna, arricchiscano l'insieme dei tipi di dati a disposizione del programmatore.

Definizione di struttura

Una **struttura** é un insieme logico di informazioni che fanno riferimento allo stesso oggetto della realtà che si intende modellare: in particolare il linguaggio permette, tramite la definizione di strutture progettate dal programmatore, di lavorare su variabili che rappresentano informazioni complesse, fornendo quindi un'astrazione migliore di quello che potrebbero fare i tipi primitivi, come `int`, `float`, ecc. Una struttura è composta da una serie di **campi** o **attributi**, ognuno dei quali può essere a sua volta una struttura o un tipo di dato primitivo.

Nel linguaggio C++ è stata introdotta la parola chiave `struct` che permette la definizione di strutture nel seguente modo:

```
struct nome_struttura{
    elenco dei campi
};
```

Riprendendo l'esempio precedente e limitandosi al problema della gestione delle date[1], si nota che sono presenti due diverse informazioni rappresentate come date, la data di nascita dello studente e la data in cui è stato dato un voto. Entrambe queste date sono però composte ognuna da tre variabili differenti, una per il giorno, una per il mese e una per l'anno.

Utilizzando la parola chiave `struct` è possibile definire invece un nuovo tipo di dato che conterrà al suo interno il giorno, il mese e l'anno, come si può vedere nel listato 7.2.

1: Le date sono un'informazione così comune che praticamente ogni linguaggio ha già presente qualche meccanismo per rappresentarle e gestirle, quindi la struttura qui definita ha il solo scopo di fornire un esempio: nei casi reali è meglio utilizzare quello che già offre il linguaggio.

```
1  struct Data{
2      int giorno;
3      int mese;
4      int anno;
5  };
```

Listing 7.2: Definizione di una struttura per rappresentare una data.

Una volta effettuata questa definizione, sarà possibile dichiarare nuovi variabili di tipo `Data` e utilizzarle in ogni parte di un programma dove è possibile usare un tipo primitivo, quindi ad esempio come parametri di una funzione, come tipo del valore di ritorno, come elementi di un vettore, ecc.

La definizione di nuove strutture viene fatta generalmente all'inizio del programma, in ambito globale, cioè al di fuori di qualsiasi funzione, `main` compreso. Questo fa sì che il nuovo tipo di dato possa essere «visto» e utilizzato in qualsiasi parte del sorgente.

Per convenzione il nome delle strutture viene indicato con la lettera maiuscola e, nel caso fosse composto, mettendo la maiuscola ad ogni parola che lo compone, come ad esempio in `LetturaContatore` .

Dichiarazione di variabili

Una volta definita una struttura, il primo utilizzo che se ne può fare è quello di definire delle variabili di quel nuovo tipo, come nell'esempio 7.3.

Listing 7.3: Dichiarazione di variabili di tipo `Data`.

```
1    Data nascita, iscrizione;
2    Data a = {19, 6, 1971};
3    nascita = a;
4    iscrizione.giorno = 10;
5    iscrizione.mese = 3;
6    iscrizione.anno = 2024;
```

Come si può vedere alla riga 1, la dichiarazione è del tutto analoga a quella di un qualsiasi altro tipo di dato primitivo: viene prima indicato il tipo, in questo caso `Data`, e, successivamente, il nome della variabile o delle variabili che si intendono dichiarare. Come nel caso di variabili dei tipi primitivi, il valore in esse contenuto non è definito. Esiste comunque la possibilità di poterne definire il valore al momento della dichiarazione, utilizzando la sintassi presente alla riga 2 dell'esempio, attraverso la quale vengono valorizzati i singoli campi, in questo caso il giorno, il mese e l'anno, inserendoli all'interno di parentesi graffe, ognuno separato dalla virgola e nell'esatto ordine in cui sono stati elencati all'interno della dichiarazione della struttura.

Per quanto riguarda l'assegnamento, il funzionamento è lo stesso già visto per i tipi primitivi, e quello che succede «dietro le quinte» è una copia del contenuto della memoria dalla variabile a destra su quella a sinistra: nel listato 7.3 alla riga 3, dopo l'assegnamento la variabile `nascita` avrà al suo interno la data 19/06/1971.

Se invece si ha la necessità di assegnare i singoli campi, perché ad esempio se ne vuole modificare uno in particolare e non tutti e

tre, è possibile indicare singolarmente i vari campi utilizzando la *dot-notation*[2] o *notazione puntata*, tramite la quale viene individuata la variabile di interesse, a cui segue un punto e il nome del campo su cui si intende operare, come descritto nelle righe dalla 4 alla 6.

In Figura 7.1 si può vedere graficamente come una struttura è un insieme di più campi, ognuno contenente un valore e che possono essere raggiunti singolarmente attraverso la notazione puntata.

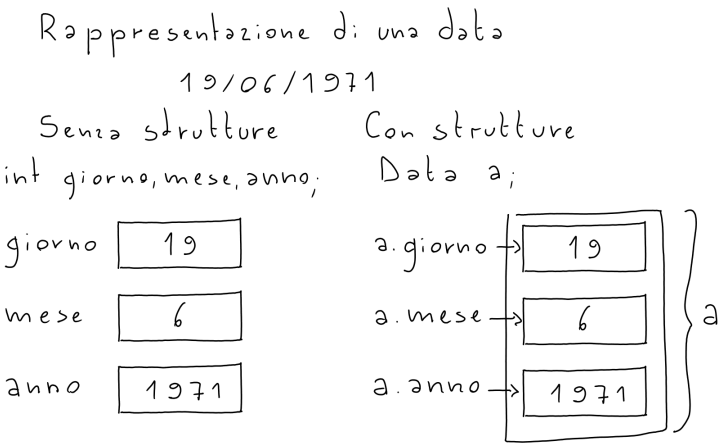


Figura 7.1.: Rappresentazione grafica di una struttura.

7.2. Operazioni elementari sulle strutture

Essendo le strutture dei tipi di dati definiti dall'utente, è ragionevole domandarsi quali operazioni possano essere compiute su variabili di questo tipo.

Operazioni di I/O

I costrutti del C++ per la lettura/scrittura da tastiera e su schermo, `cin` e `cout`, rendono molto comode queste due operazioni, ma non possono funzionare sulle strutture definite dal programmatore. Questo significa che volendo leggere o scrivere una data, riprendendo l'esempio già visto, il procedimento corretto è quello mostrato nel listato 7.4:

```
1    std::cout << "Inserisci da data di iscrizione: ";
2    std::cin >> iscrizione; //ERRORE
3    std::cin >> iscrizione.giorno;
4    std::cin >> iscrizione.mese;
5    std::cin >> iscrizione.anno;
```

Listing 7.4: Lettura di variabili di tipo `Data`.

Sebbene la riga 2 potrebbe sembrare logica da un punto di sintattico, non può funzionare ed emetterebbe un errore in fase di compilazione, poiché al momento della scrittura del codice di

2: Questo tipo di notazione era già stata presentata lavorando su `vector` e sulle `string`: in quel caso l'ambito di utilizzo era quello di utilizzare dei *metodi*, ma l'idea di fare riferimento a una variabile indicata prima del punto è la stessa.

libreria che permette alla `cin` di funzionare, erano noti solo i tipi primitivi, non certo le infinite possibili strutture definite dai programmatori. Il problema si risolve facilmente ricorrendo alla dot-notation e lavorando sui campi che compongono una data, che essendo di tipo `int`, possono essere gestiti dalle primitive di input.

Se si volesse personalizzare l'input, ad esempio nel caso delle date, per permettere direttamente l'inserimento di qualcosa di più *user-friendly*, come ad esempio 19/06/1971, senza dover ricorrere a tre inserimenti di interi distinti, sarebbe il programmatore a doversi fare carico della lettura della stringa contenente anche gli *slash*, per poi farne il *parsing*, suddividerla nelle tre parti e inserirle negli appositi campi della variabile di tipo `Data`[3].

Per quanto riguarda l'output, il discorso è analogo e quindi non è possibile inviare direttamente una variabile di tipo `Data` direttamente a `cout`, ma si può procedere come nel modo indicato nel seguente listato.

```
1    std::cout << "Data di iscrizione: " <<
2        iscrizione.giorno << "/" <<
3        iscrizione.mese << "/" <<
4        iscrizione.anno << std::endl;
```

Listing 7.5: Scrittura di variabili di tipo `Data`.

che potrebbe produrre come output

```
Data di iscrizione: 10/3/2024
```

Va notato che qualsiasi *abbellimento* dell'output, che in questo caso potrebbe essere quello di mettere uno zero davanti ai giorni o ai mesi minori di 10, deve essere gestita dal programmatore, poichè, di nuovo, i meccanismi «standard» lavorano sugli interi e nulla sanno sulla rappresentazione delle date.

3: Utilizzando una caratteristica del linguaggio, l'*overloading* degli operatori, sarebbe possibile fare in modo di adattare `cin` per farlo funzionare sulle date, ma questo argomento non verrà trattato.

Operatori aritmetici

Come dovrebbe risultare ovvio, gli operatori di somma, differenza, moltiplicazione, divisione e resto, non possono essere utilizzati per le strutture definite dall'utente. Nonostante questo, come già visto per le `string` e l'operatore +, esistono dei casi nei quali alcune di queste operazioni potrebbero assumere significato anche se gli operandi non sono numeri. Ad esempio, per le date potrebbe aver senso *sommare* a una data un numero di giorni, per vedere a quale nuova data si arriva: in questi casi sarà il programmatore a definire delle funzioni che svolgeranno i compiti desiderati[4].

4: Come già indicato, tramite *overloading* degli operatori, sarebbe possibile fare in modo di *ridefinire* il comportamento degli operatori aritmetici e anche quelli di confronto per adattarli alle strutture definite dal programmatore.

Operatori di confronto

Anche per gli operatori di confronto, come `<`, `<=`, `==`, ecc. vale lo stesso discorso fatto per gli operatori aritmetici: non essendo definiti, ma essendoci molte situazioni nelle quali è utile confrontare tra di loro delle variabili di tipo struttura, sarà compito del programmatore definire il significato del confronto, come si può vedere nel listato 7.6, dove vengono confrontate due date, `d1` e `d2`, per stabilire quale viene prima e quale viene dopo.

Listing 7.6: Confronto tra variabili di tipo `Data`.

```
Data d1 = {19, 6, 1971},
d2 = {22, 6, 1974};

if (d1.anno < d2.anno) {
    std::cout << "d1 < d2";
}
else if (d1.anno > d2.anno) {
    std::cout << "d1 > d2";
}
else {
    if (d1.mese < d2.mese) {
        std::cout << "d1 < d2";
    }
    else if (d1.mese > d2.mese){
        std::cout << "d1 > d2";
    }
    else {
        if (d1.giorno < d2.giorno) {
            std::cout << "d1 < d2";
        }
        else if (d1.giorno > d2.giorno){
            std::cout << "d1 > d2";
        }
        else {
            std::cout << "d1 = d2";
        }
    }
}
```

Il codice 7.6 dimostra in modo evidente come un'operazione che, con i tipi primitivi, richiederebbe poche linee utilizzando gli operatori di confronto, con oltretutto il vantaggio della leggibilità, in questo caso diventa molto più complessa e di difficile lettura. Si vedrà come porre parzialmente rimedio a questo problema utilizzando le funzioni, per migliorare in questo modo la leggibilità del codice.

7.3. Vettori e strutture

Dovendo rappresentare un numero elevato e non necessariamente noto a priori di variabili di tipo struttura, bisogna ricorrere all'utilizzo dei vettori, come già si era visto nel caso di variabili di tipo primitivo.

Anche in questo caso è possibile ricorrere ai vettori nativi o ai `vector`, lasciando la scelta a considerazioni legate al contesto del problema e all'utilizzo che si vuol fare del vettore.

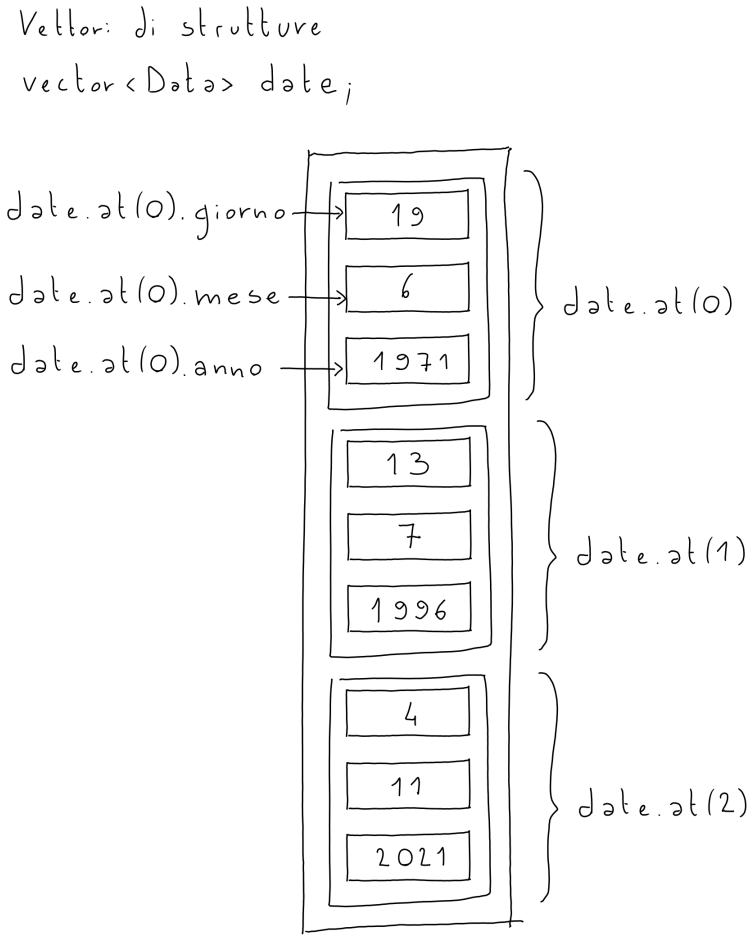


Figura 7.2.: Vettori di strutture.

Nel prosieguo del testo si preferirà l'utilizzo della classe `vector`, per i vantaggi che offre rispetto ai vettori nativi, come indicato alla sezione Sezione 5.3.

Il comportamento di un `vector` di un qualsiasi tipo di struttura sarà identico a quello di un `vector` di tipi primitivi, come `int` o `float`: sarà, ad esempio, possibile individuare il singolo elemento utilizzando il metodo `at` e passandogli l'indice desiderato, come mostrato in Figura 7.2.

Da un punto di vista della scrittura del codice non cambia nulla, l'aspetto importante da tener presente è che, una volta «recuperato»

l'elemento tramite l'indice, quella è una variabile di tipo struttura e quindi bisognerà utilizzare la *dot-notation* per accedere ai campi di interesse.

Listing 7.7: Esempio di dichiarazione e utilizzo di un `vector` di date.

```
1    std::vector<Data> date;
2    Data d;
3    for (int i = 0; i < 10; ++i) {
4        std::cout << "Inserisci una data: ";
5        std::cin >> d.giorno;
6        std::cin >> d.mese;
7        std::cin >> d.anno;
8        date.push_back(d);
9    }
10   std::cout << date.at(3).giorno << "/" <<
11       date.at(3).mese << "/" <<
12       date.at(3).anno << std::endl;
```

Come si vede nel listato 7.7, alla prima riga viene dichiarato un vettore di date, che verrà poi riempito dall'utente con 10 date, utilizzando la variabile temporanea `d` per inserire i dati forniti dall'utente e, alla riga 8, aggiungere la nuova data nel vettore *date*.

Le successive righe del programma mostrano che, scelto un qualsiasi elemento, in questo esempio il quarto elemento, questo possa essere trattato come una singola variabile, usando dopo il metodo `at` la *dot-notation* e indicando i campi che interessano.

7.4. Strutture composte

Siccome le strutture definiscono nuovi tipi di dati, una struttura, una volta definita, può essere usata a sua volta come attributo di un'altra struttura e questa cosa può essere estesa *ricorsivamente* a quanti livelli si desidera, sempre con l'attenzione che un numero troppo alto di strutture all'interno di altre strutture è probabilmente indice di una progettazione non ottimale.

Ritornando all'esempio presentato all'inizio del capitolo nel listato 7.1, si potrebbe raggiungere un risultato decisamente migliore attraverso la definizione delle strutture `Data`, `Valutazione` e `Studente`, come mostrato nel listato 7.8:

Listing 7.8: Esempio delle informazioni dei voti e degli studenti utilizzando delle strutture composte.

```
1    struct Data{
2        int giorno;
3        int mese;
4        int anno;
5    };
6
7    struct Valutazione {
8        float voto;
```

```
9         std::string materia;
10        Data data;
11    };
12
13    struct Studente {
14        std::string nome;
15        std::string cognome;
16        Data nascita;
17        int classe;
18        char sezione;
19        std::vector <Valutazione> valutazioni;
20    };
```

In questa rappresentazione, le informazioni risultano sicuramente meglio organizzate, in quanto ogni studente sarà rappresentato da un'unica variabile di tipo `Studente`, contenente al suo interno i campi che rappresentano i dati anagrafici e le sue valutazioni.

Come si può vedere la struttura `Valutazione` contiene al suo interno un campo di tipo `Data`, dove la struttura `Studente` contiene sia un campo di tipo `Data` che un vettore di variabili di tipo `Valutazione`.

Supponendo di avere una variabile `s` di tipo `Studente` e una variabile `v` di tipo valutazione, aggiungere una nuova valutazione allo studente richiederà una scrittura come:

```
s.valutazioni.push_back(v);
```

che è chiaramente più leggibile di quello che invece si sarebbe dovuto scrivere avendo definito i dati come nel listato 7.1.

7.5. Funzioni e strutture

Come già approfondito nel Capitolo 4 (Funzioni), una funzione può ricevere parametri di qualsiasi tipo, per copia o per riferimento e può ritornare un valore, sempre indicandone il tipo.

Essendo le strutture un nuovo tipo di dato, ne consegue che nelle funzioni verranno usate le stesse regole già viste in precedenza per i tipi primitivi, indicando quindi prima il tipo del parametro, che potrà essere una struttura come ad esempio `Data` e successivamente il suo nome. Lo stesso discorso varrà per il tipo di ritorno, che potrà essere una struttura.

Anche in questo dato l'utilizzo di strutture anziché di variabili di tipi primitivi, migliorerà la leggibilità del codice, rendendolo più «compatto» e facile da comprendere.

Per fare un esempio, nel listato 7.9 si può vedere come viene definita una funzione per il confronto tra date, che migliora quanto visto nel listato 7.6. Questa funzione ritornerà un numero negativo per

indicare che la prima data è precedente alla seconda, ritornerà un numero positivo per indicare che la prima data è successiva alla seconda, oppure zero per indicare che sono la stessa data[5].

5: Questo modo di utilizzare un valore di ritorno intero per indicare la relazione d'ordine tra due parametri è piuttosto comune e viene utilizzato ad esempio in C e in Java quando avvengono operazioni di confronto tra stringhe.

Listing 7.9: Funzione per il confronto tra variabili di tipo `Data`.

```
int confronta_date(const Data &d1, const Data &d2) {
    if (d1.anno - d2.anno != 0)
        return d1.anno - d2.anno;
    if (d1.mese - d2.mese != 0)
        return d1.mese - d2.mese;
    return d1.giorno - d2.giorno;
}
```

Va notato che, come già indicato nella Sezione 6.4 (Stringhe e funzioni), anche per le strutture il passaggio dei parametri conviene avvenga per riferimento, indicandolo come `const` laddove si vuole indicare al compilatore la volontà che il parametro non venga modificato all'interno del corpo della funzione, come succede nel listato 7.9.

Infine, nulla cambia rispetto a quanto già visto, se si volesse passare come parametro di una funzione un vettore di strutture e si volesse far ritornare alla funzione un vettore di strutture. Per quanto riguarda il passaggio di parametro, va solo tenuto presente che, se non vuole modificare il vettore, si sceglierà il passaggio per riferimento costante, mentre il caso in cui lo si voglia modificare prevederà il passaggio per riferimento (il passaggio per copia, per i motivi già detti, non verrà mai utilizzato).

Per quanto riguarda il valore di ritorno invece, è del tutto lecito ritornare un vettore di elementi di una qualche tipo di struttura, che verrà poi assegnato alla variabile opportuna.

Per fornire un paio di esempi di *firme* di funzioni che utilizzano parametri e valori di ritorno che sono vettori di strutture, si guardi il listato 7.10

Listing 7.10: Esempi di funzioni che hanno come parametro o ritornano un vettore di strutture.

```
int date_maggiori_di(const std::vector<Data> &date,
Data d);

void rimuovi_doppi(std::vector<Data> &date);

std::vector<Data> crea_date(int mese, int anno);
```

La prima funzione, `date_maggiori_di` ha lo scopo di contare quante delle date contenute all'interno al vettore `date` sono maggiori di una certa data `d`, passata come secondo parametro. In questo caso il vettore di date viene passato come riferimento costante, in quanto non c'è necessità di modificarlo.

La seconda funzione, `rimuovi_doppi`, elimina eventuali date doppie presenti all'interno del vettore `date`: questa operazione può

modificare il contenuto del vettore, che quindi viene passato come riferimento, stavolta non costante.

L'ultima funzione, `crea_date`, ha lo scopo di creare tutte le date consecutive di un certo mese e di un certo anno, passati entrambi come parametri. Se ad esempio il mese fosse il 2 (febbraio) e l'anno il 2024, la funzione dovrebbe creare 29 date, dal 01/02/2024 al 29/02/2024. Il modo più comodo per poi ritornare le date al chiamante, sarebbe quello di «impacchettarle» all'interno di un vettore e ritornarlo, come si vede dalla firma della funzione.

7.6. Esercizi

Studi

In questi studi viene lasciata ampia discrezionalità di implementazione, poichè lo scopo è soprattutto quello di verificare la comprensione delle caratteristiche sintattiche delle strutture e fare un po' di esercizio. Per la parte in cui viene richiesta l'implementazione di funzioni, si scelgano parametri e valori di ritorno nel modo che si ritiene più opportuno.

1. Definire una struttura adatta a rappresentare il biglietto di un concerto. Scrivere successivamente un programma per far inserire all'utente i dati di un concerto all'interno di una variabile e successivamente stamparla per verificare la correttezza di quanto fatto.
2. Definire una struttura adatta a rappresentare le caratteristiche di un computer. Scrivere successivamente un programma per far inserire all'utente i dati di un computer all'interno di una variabile e poi stamparla per verificare la correttezza di quanto fatto.
3. Definire una struttura adatta a rappresentare un'automobile. Scrivere successivamente un programma per far inserire all'utente i dati di un'automobile all'interno di una variabile e poi stamparla per verificare la correttezza di quanto fatto.
4. Definire una struttura adatta a rappresentare una città. Scrivere successivamente un programma per far inserire all'utente i dati di una città all'interno di una variabile e poi stamparla per verificare la correttezza di quanto fatto.
5. Definire una struttura adatta a rappresentare il biglietto di un viaggio aereo. Scrivere successivamente un programma per far inserire all'utente i dati di un viaggio all'interno di una variabile e poi stamparla per verificare la correttezza di quanto fatto.
6. Definire una struttura adatta a rappresentare una carta d'identità. Scrivere successivamente un programma per far inserire all'utente i dati di una carta d'identità all'interno di una variabile e poi stamparla per verificare la correttezza di quanto fatto.
7. Definire le strutture adatte a rappresentare una squadra di calcio e i calciatori che la compongono. Scrivere successivamente un programma che inizializzi da codice i dati di una squadra all'interno di una variabile e poi la stampi per verificare la correttezza di quanto fatto.
8. Definire le strutture adatte a rappresentare una classe, i docenti e gli studenti che la compongono. Scrivere successivamente un programma che inizializzi da codice i dati di una classe all'interno di una variabile e poi la stampi per verificare la correttezza di quanto fatto.
9. Definire le strutture adatte a rappresentare un viaggio composto da più tappe. Scrivere successivamente un programma che inizializzi da codice i dati di un viaggio all'interno di una variabile e successivamente la stampi per verificare la correttezza di quanto fatto.
10. Data la struttura che rappresenta una squadra di calcio vista negli esercizi precedenti, scrivere una funzione `stampa_squadra` che stampi le informazioni sulla squadra e sui giocatori che la compongono.
11. Data la struttura che rappresenta una squadra di calcio vista negli esercizi precedenti, scrivere una funzione `stampa_ruolo` che stampi l'elenco dei giocatori che giocano in un certo ruolo passato come parametro alla funzione.
12. Data la struttura che rappresenta una squadra di calcio vista negli esercizi precedenti, scrivere una funzione `goal_segnati` che ritorni il numero di goal segnati dalla squadra, ottenuti come somma dei goal segnati da ogni singolo giocatore.

13. Data la struttura che rappresenta una squadra di calcio vista negli esercizi precedenti, scrivere una funzione `capocannoniere` che ritorna il giocatore o i giocatori che hanno segnato il maggior numero di reti.
14. Data la struttura che rappresenta una classe vista negli esercizi precedenti, scrivere una funzione `stampa_studenti` che stampi l'elenco degli studenti della classe, senza nessun particolare ordine.
15. Data la struttura che rappresenta una classe vista negli esercizi precedenti, scrivere una funzione `stampa_docenti` che stampi l'elenco dei docenti della classe, senza nessun particolare ordine.
16. Data la struttura che rappresenta una classe vista negli esercizi precedenti, scrivere una funzione `scegli_studente` che ritorni uno studente della classe scelto a caso.
17. Data la struttura che rappresenta una classe vista negli esercizi precedenti, scrivere una funzione `estrai_studenti` che ritorni una lista di studenti, eventualmente vuota, contenente tutti gli studenti il cui cognome inizia con una lettera, passata come parametro alla funzione.
18. Data la struttura che rappresenta una classe vista negli esercizi precedenti, scrivere una funzione `docente_materia` che ritorni il docente che insegna la materia passata come parametro alla funzione.
19. Data la struttura che rappresenta una classe vista negli esercizi precedenti, scrivere una funzione `stampa_studenti` che stampi l'elenco degli studenti della classe, senza nessun particolare ordine.
20. Si implementino le funzioni indicate nel listato 7.10, scrivendo successivamente dei pezzi di programma che permettano di verificarne la correttezza.

Esercizi

1. Nelle gare delle Olimpiadi di Informatica, ogni partecipante deve provare a risolvere un set di problemi, ognuno dei quali è caratterizzato da un titolo, composto da una parola soltanto, e dal proprio valore, un numero intero che indica il punteggio massimo che può essere ottenuto per quell'esercizio. I partecipanti, man mano che sottopongono le proprie soluzioni al sistema di gara, ricevono un punteggio per il problema che hanno provato a risolvere, che va da zero (soluzione completamente sbagliata) fino al valore di quel problema. Il punteggio totale per un partecipante sarà poi calcolato sommando il massimo punteggio ottenuto per ogni singolo problema.
   Scrivere un programma che permetta di simulare l'andamento di una gara, implementando le seguenti funzionalità:
   - inserimento dei problemi di gara, con titolo e valore, fatto una sola volta all'inizio del programma. Il numero di problemi deve poter essere scelto dall'utente.
   - inserimento di un nuovo partecipante: un partecipante è individuato dal nome, da cognome, dalla sede territoriale dove gareggia, che è una stringa, ad esempio potrebbe essere LOM4 o EMI3, e dall'elenco dei punteggi fatti per ogni singolo problema
   - aggiornamento del punteggio di un partecipante su uno specifico problema: se il punteggio fosse inferiore a quello già presente per quel problema, il punteggio non verrà aggiornato. Se il punteggio fosse maggiore del valore del problema, il punteggio varrà il valore del problema
   - stampa del punteggio di un partecipante, individuato dal cognome (si può supporre che i cognomi siano unici)

- stampa delle informazioni del partecipante (o dei partecipanti) che ha il punteggio totale maggiore, con le indicazioni dei singoli punteggi di ogni problema
- stampa l'elenco dei partecipanti di una sede territoriale

2. In un sistema domotico di gestione dell'illuminazione, ogni lampada è caratterizzata dalla propria potenza in Watt, dalla quantità di illuminazione, che è un intero compreso tra 0 e 100 (100 a piena potenza, 0 è come se fosse spenta, 50 è alla metà della propria potenza) e dal colore, che può essere "giallo", "bianco" o "blu" e infine da un nome che le viene assegnato quando viene aggiunta al sistema. Il sistema deve essere in grado di gestire al massimo 100 lampade. Ogni lampada può essere accesa o spenta individualmente. Scrivere una o più strutture per rappresentare i dati di una singola lampada e del sistema e successivamente scrivere un programma che permetta, tramite un menù testuale, di svolgere le seguenti operazioni:
   - aggiungere una nuova lampada al sistema, assegnandole un nome
   - modificare la potenza di una lampada, o assegnandole un nuovo valore, oppure aumentandola o diminuendola del 10%
   - modificare il colore di una lampada
   - calcolare la potenza totale attuale del sistema. La potenza del sistema si ottiene con la semplice formula

     $$P_{tot} = \sum_{1}^{N} P_i \times \frac{Q_i}{100}$$

     dove P_i è la potenza della lampada *i*-esima e Q_i è la quantità di illuminazione della lampada *i*-esima. Le lampade spente ovviamente non aggiungono potenza al sistema.
   - spegnere o accendere ogni singola lampada individualmente oppure spegnere o accendere ogni lampada del sistema
   - rimuovere una lampada dal sistema
3. Una ditta che gestisce la lettura dei contatori di impianti elettrici civili ha necessità di registrare per ogni *lettura* il codice dell'operatore (alfanumerico di 10 caratteri), il codice dell'impianto (alfanumerico di 10 caratteri), la data di lettura e il numero letto sul contatore, che rappresenta i kWh consumati fino all'istante di lettura (ad esempio 12034.32). Scrivere una o più strutture per rappresentare i dati delle letture e successivamente scrivere un programma che permetta, tramite un menù testuale, di svolgere le seguenti operazioni:
   - aggiungere una nuova lettura
   - mostrare tutte le letture fatte da un certo operatore identificato con il suo codice
   - mostrare tutte le letture relative a uno stesso impianto identificato con il suo codice
   - calcolare a quanto ammonta l'ultima bolletta di un certo impianto, dato il codice dell'impianto e il costo per Kwh. Per il calcolo bisogna utilizzare le ultime due letture e fare la differenza tra i due valori di KWh consumati, per mostrare una bolletta simile a quella mostrata in questo esempio:

     ```
     Codice impianto: AA12345FF0
     Bolletta del periodo 01/01/2023 - 31/03/2023
     KWh utilizzati: 123.43
     Costo al KWh: 0.23 euro
     Totale: 28.39 euro
     ```

Se non ci fosse nessuna lettura per quel codice o se ce ne fosse solo una, il programma deve segnalare l'errore.

- mostrare tutte le letture fatte in un certo periodo di tempo, espresso da una data d'inizio e da una data di fine.

File 8.

Introduzione

Tutti i programmi scritti finora hanno seguito un modello generale, nel quale l'utente forniva dei dati in input (numeri o stringhe), eseguiva un qualche tipo di elaborazione e, successivamente, i valori calcolati venivano mostrati a video, dopodiché il programma terminava.

Il limite di questo approccio è che ogni informazione, sia in input che in output, viene persa alla chiusura del programma: quindi, volendo ad esempio riottenere dei risultati calcolati in precedenza, è necessario rieseguire il programma, fornendo gli stessi input e calcolare nuovamente il risultato delle operazioni di interesse. Chiaramente per moltissimi programmi questo non è accettabile e quindi lo scopo di questo capitolo sarà quello di vedere come rendere *persistenti*[1] le informazioni gestite da un programma.

In questo capitolo ci si concentrerà esclusivamente su file che contengono dati: dovrebbe risultare comunque evidente che, siccome ogni informazione viene memorizzata su file, anche i programmi eseguibili sono dei file: quindi tutti gli eseguibili prodotti fin qua, piuttosto che l'ambiente di sviluppo o il browser che si utilizza per navigare nel Web, sono memorizzati nel *file system* sotto forma di file.

1: Un'informazione si dice *persistente* quando sopravvive alla chiusura del programma che la gestisce e può essere recuperata ad una successiva esecuzione del programma.

8.1. I file

Le informazioni su un computer, di qualsiasi tipo esse siano, vengono rappresentate attraverso codici binari, sequenze di 0 e 1. Chiaramente queste sequenze devono essere organizzate in un qualche modo, per poter essere memorizzate, lette e modificate e il modo in cui questo viene fatto è quello di raggrupparle all'interno di contenitori di informazioni chiamati *file*.

Definizione di file

Un **file** è un contenitore di informazioni, in forma binaria, che rappresentano un oggetto all'interno di un sistema computerizzato. Qualsiasi informazione viene memorizzata all'interno di un file per permettere di recuperarla in un momento successivo alla sua creazione, implementando in questo modo un meccani-

smo di accesso ai dati uniforme e indipendente dal supporto dove fisicamente avviene la memorizzazione.

I sistemi operativi hanno una parte dedicata a gestire tutte le problematiche relative alla memorizzazione di file, che viene chiamata *file system*, e che astrae la complessità[2] legata alla gestione della parte elettronica e/o meccanica e mostra al programmatore solo dei metodi di accesso universali e indipendenti dai dettagli tecnici. Grazie a questo è possibile solo concentrarsi su alcuni aspetti «logici», che vengono elencati nei seguenti paragrafi.

2: Ogni singolo bit deve essere memorizzato su un qualche dispositivo elettronico, i vari bit devono essere organizzati un gruppi più grandi che rappresentano informazioni strutturate, i supporti fisici possono essere dischi con tecnologia elettromagnetica, chiavette USB, schede SSD e altro: questo elenco dovrebbe essere sufficiente per far intuire l'enorme complessità che, come si vedrà, chi programma non ha bisogno di conoscere.

Nome del file

Il nome di un file è probabilmente il singolo attributo più importante, sia per un utente, che per un programmatore. Da un punto di vista dell'utente permette di individuare l'oggetto di interesse (un documento di testo, un'immagine, un video, ...), per un programmatore è invece il modo con cui, all'interno di un programma, si può fare riferimento al file su cui si desidera operare. Esempi di nomi di file sono mostrati nella sezione 8.1

Listing 8.1: Esempi di nomi di file

```
fig1.png
fig2.png
footing
footing_backtracking.cpp
footing.cpp
footing_djikstra.cpp
footing.pdf
footing.tar.gz
```

Come si può notare il nome è spesso[3] composto da una prima parte[4], un punto e successivamente una seconda parte, normalmente chiamata *estensione*. Il nome è l'insieme di tutte queste parti ed è con il nome completo che dovrà essere individuato un file all'interno di un programma.

3: Ma non necessariamente, infatti si può notare che un nome non contiene nessun punto, mentre un altro ne contiene due.

4: Al momento della scrittura di questi appunti, l'interfaccia grafica del sistema operativo Windows mostra per default solo la prima parte del nome, si consiglia di disabilitare questo comportamento per vedere sempre il nome completo.

Un'altra cosa importante è che il nome da solo non è detto che individui univocamente un file, poichè l'organizzazione dei file system permette di avere più file con lo stesso nome, l'importante è che risiedano in *cartelle* diverse, che a loro volta possono contenere altre cartelle, ricorsivamente, generando una gerarchia ad albero, come quella che si vede nella Figura 8.1.

Data questa organizzazione, un file viene univocamente identificato dal suo nome e dal percorso che serve a raggiungerlo partendo dalla radice del file system fino ad arrivare alla cartella che lo contiene. Due esempi, relativi rispettivamente al S.O. Linux e a Windows, sono i seguenti:

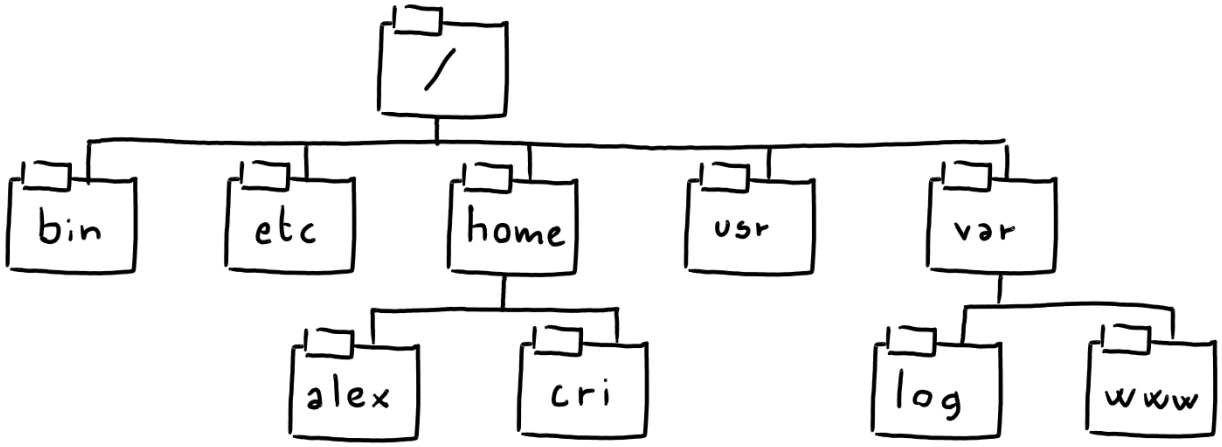


Figura 8.1.: Esempio dell'albero delle cartelle in un sistema Linux.

Listing 8.2: Percorsi assoluti

```
1 /home/alex/first_project/footing.cpp (linux)
2
3 c:\users\alex\first_project\footing.cpp (windows)
4
```

L'insieme del percorso e del nome del file si definisce percorso **assoluto**. Spesso nei programmi si preferisce utilizzare un percorso **relativo**, dove per relativo si intende rispetto alla posizione dell'eseguibile: se ad esempio l'eseguibile si trovasse in `/home/alex/first_project` e il file `dati.txt` si trovasse nella cartella `/home/alex/first_project/files`, allora il suo percorso relativo sarebbe `./files/dati.txt`.

Va infine notato che sui sistemi operativi *Unix-like*, come ad esempio Linux, i nomi dei file sono *case-sensitive*, cioè, ad esempio, `hello.txt` è differente da `Hello.txt`, mentre sui sistemi operativi Windows i nomi dei file sono *case-insensitive*, cioè `hello.txt` e `Hello.txt` sono lo stesso file[5]. Come regola generale si cercherà quindi di scegliere nomi che contengano tutte lettere minuscole, per evitare ogni tipo di problema con programmi che dovessero essere eseguiti in ambienti diversi.

5: Il problema è un po' più articolato di così, ma per gli scopi di questo testo, questa informazione è sufficiente per evitare di fare errori.

Tipo del file

Se il nome identifica un file, il suo contenuto è ciò che definisce l'informazione che rappresenta. Come detto all'inizio del capitolo, il contenuto è sempre una serie di bit, però il tipo del file permette di dare la corretta interpretazione al suo contenuto e, quindi, di poterlo elaborare all'interno di un programma per ottenere ciò che si desidera. A solo titolo di esempio verranno qua elencati alcuni tipi di file, indicandone l'estensione tipica:

- `jpg`: contengono immagini, pensati per rappresentare fotografie attraverso una compressione `lossy`, che può essere impostata per ottenere una qualità più o meno buona
- `png`: contengono immagini, in questo caso funzionano meglio per immagini *sintetiche*, anche in questo caso utilizzano algoritmi di compressione `lossy` e possono supportare la trasparenza
- `docx, odt`: contengono documenti editabili con una vasta serie di attributi, come *font*, *dimensione* dei caratteri, *stili*, *dimensioni* dei margini e della pagina, ecc.
- `mp4`: contengono filmati compressi
- `cpp`: contengono codice sorgente C++ in forma testuale
- `txt`: contengono del testo libero o organizzato in dipendenza al loro scopo.
  Si ricorda ancora che l'estensione è solo una convenzione che permette di individuare rapidamente il tipo di file[6], ma è il suo contenuto a definirlo: quindi rinominare un file `.txt` in un file `.jpg` **non** lo renderà un'immagine.

6: Nel caso dell'interfaccia grafica di Windows, l'estensione permette in effetti di mostrare l'icona grafica opportuna e cambiando l'estensione l'icona cambierà, portando il più delle volte all'attivazione di un programma di gestione non corretto.

File testuali e binari

Sebbene tutti i file non siano altro che sequenze di zero e uno, dal punto di vista dei programmatori una possibile distinzione è quella tra *file di testo* e *file binari*.

Come indica la definizione, i file di testo sono dei file che al loro interno contengono dei caratteri che possono essere letti e scritti da una persona utilizzando un semplice editor di testo.

Esempi di tipici file di testo sono i file di configurazione, quelli di *log*, alcune forme di esportazione dei dati, come i file in formato `csv` (Comma Separated Values), i codici sorgenti di un programma, lo stesso file da cui questo documento viene generato[7]. Ognuno di questi file ha poi generalmente una sua sintassi interna che lo rende «comprensibile» ai programmi che dovranno usarlo.

7: Questo testo è scritto in LaTeX, un sistema per la creazione di documenti, che a partire da un file testuale scritto seguente certe regole, compone un documento tipograficamente professionale.

Viceversa, i file binari contengono al proprio interno qualsiasi insieme di byte e, come tali, non sono interpretabili direttamente dagli umani, ma hanno bisogno di essere gestiti da programmi specifici pensati per renderli fruibili, come, ad esempio, i visualizzatori di immagini, i player audio, i programmi di grafica, ecc. Il tentativo di apertura di questi file con un generico editor di testo, come ad esempio il blocco note di Windows o il programma testuale `nano` di Linux, porterà nella migliore delle ipotesi a non visualizzare nulla, nella peggiore un insieme incomprensibile di simboli più o meno strani.

In questo capitolo ci si occuperà esclusivamente dei file testuali, in quanto più semplici da capire e utilizzare per le esigenze dei problemi che si vorranno risolvere.

Altre caratteristiche

Oltre a quelle elencate, un file presenta altre caratteristiche generali che, sebbene non rilevanti ai fini della scrittura dei programmi che verranno presentati nel seguito, è bene conoscere.

Alcune delle principali sono:

- **dimensioni**: è la quantità di spazio che occupa su disco, generalmente espressa in *byte*. Per come è organizzato il *file system*, non è detto che coincida esattamente con il numero di byte effettivi presenti nel file
- **proprietario**: i *file system* moderni associano a ogni file un qualche concetto di «proprietà». Senza entrare nel dettaglio, per un programmatore è importante sapere che gestire un file di cui non si è proprietari potrebbe portare al fallimento dell'operazione
- **permessi**: sono associati al concetto di proprietà ed esprimono cosa può essere fatto su un certo file, anche a seconda della relazione di proprietà su di esso. Ai fini del seguente capitolo, i permessi che interessano sono quelli di lettura/scrittura: in generale, lavorare su un file che è in proprio possesso, darà accesso sia alle operazioni di lettura che a quelle di scrittura
- **data di creazione**: indica quando è stato creato il file, dato che viene preso dall'orologio di sistema. Oltre a questa data esistono anche quella di modifica e di ultimo accesso, ma la data di creazione sarà rilevante per verificare se le operazioni fatte dal programma hanno in effetti creato nuovi file nel momento in cui sono state eseguite

Tutte le caratteristiche elencate sopra possono essere individuate all'interno del listato 8.3, che mostra un esempio di informazioni prodotte dal comando `ls -l` di Linux[8] per elencare gli attributi dei file presenti nella cartella dove si sta eseguendo il comando.

8: Il comando equivalente in Windows è `dir`

Listing 8.3: Esempio delle caratteristiche dei file contenuti in una cartella nel S.O. Linux.

```
-rwxr-xr-x 1 alex alex   5163 apr 16 22:12 ale.jpg
drwxr-xr-x 3 alex alex   4096 ott 21  2022 go
-rw-r--r-- 1 alex alex   1075 mar 13  2021 LICENSE
drwxr-xr-x 3 alex alex   4096 feb 19 22:18 prova_ddev
-rw-r--r-- 1 alex alex   6101 mar 13  2021 README.md
-rw-r--r-- 1 alex alex    207 nov  7 18:02 texput.log
-rw-r--r-- 1 alex alex   2898 apr 13 09:11 world.pdf
-rw-r--r-- 1 alex alex 159090 feb 23  2023 world.png
```

8.2. Gestione dei file

La gestione dei file testuali in C++, come nella maggior parte dei linguaggi, prevede che vengano fatte una serie di operazioni, sempre nello stesso ordine, come mostrato in Figura 8.2.

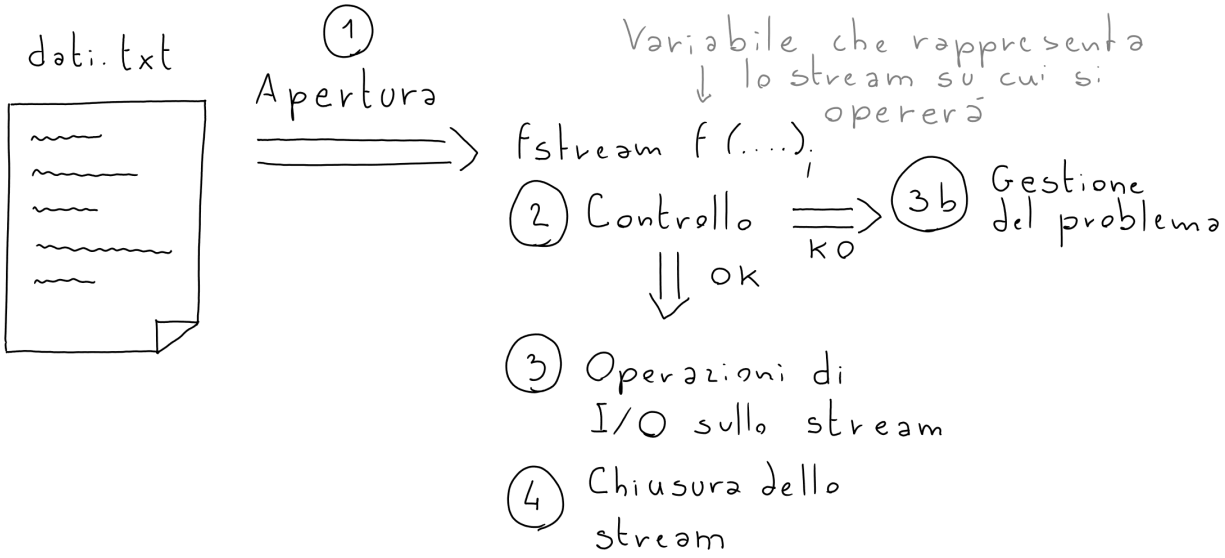


Figura 8.2.: Sequenza di operazioni per la gestione dei file

Apertura dello stream

In generale si vedrà che in C++ i file testuali vengono trattati come gli *stream* che usualmente vengono utilizzati per l'interazione con l'utente, cioè `std::cin` e `std::cout`. A differenza di quest'ultimi, non vengono automaticamente aperti all'avvio del programma, ma hanno bisogno di istruzioni apposite, come quelle viste nel frammento di programma presente nel listato 8.4:

Listing 8.4: Apertura di file

```
1  #include <iostream>
2  #include <fstream>
3
4  int main() {
5      std::ifstream in("input.txt");
6      std::ofstream out("output.txt");
```

La prima cosa da notare è che, come `std::cin` e `std::cout` hanno bisogno dell'inclusione del file di libreria `iostream`, per la gestione dei file è necessario includere il file di libreria `fstream`.

Volendo poi procedere con l'apertura del file, o dei file, verrà indicato se si vuole aprire il file in lettura o in scrittura[9].

Queste operazioni avvengono usando le istruzioni alla riga 5 per quanto riguarda l'apertura in lettura e alla riga 6 per l'apertura in scrittura.

9: É anche possibile aprire lo stesso file contemporaneamente in lettura/scrittura o in altre modalità, che però non verranno trattate in quanto inutilmente complesse rispetto a quello che si vorrà fare.

Un file viene aperto in lettura quando il programma ha necessità di leggere le informazioni al suo interno per poterle caricare nelle variabili opportune, mentre viene aperto in scrittura quando il programma deve memorizzare su file le informazioni che elabora durante la sua esecuzione.

Come si può vedere alle righe 5 e 6, la scrittura prevede che venga utilizzata una notazione che assomiglia alle funzioni già incontrate[10], dove il parametro è il nome del file che si è interessati ad aprire, sotto forma di stringa. Come già indicato alla Sezione 2, il nome può essere indicato utilizzando un percorso relativo o assoluto: nei codici presentati si preferirà sempre utilizzare un percorso relativo, per garantire il funzionamento indipendentemente dalla posizione precisa dove verrà eseguito il programma. La cosa più semplice è quella di indicare solo il nome del file, stando attenti a fare in modo che eseguibile e file si trovino nella stessa cartella[11].

10: Tecnicamente è un *costruttore*.

L'apertura in qualche modo fa un *binding*, un collegamento, tra il nome del file che ha nel *file system* e il nome della variabile che verrà successivamente utilizzato nel programma. Nell'esempio mostrato il file `input.txt` viene associato alla variabile `in` e quindi nelle successive righe per far riferimento a tutte le operazioni su quel file verrà utilizzata la variabile `in` e non più il nome del file.

11: Ambienti di sviluppo diversi organizzano in maniera differente il processo di esecuzione di un programma, quindi in fase di sviluppo sarà necessario individuare dove memorizzare il file per permettere l'esecuzione corretta del programma.

Controllo sull'apertura

A differenza degli *stream* di I/O, che risultano sempre definiti, quando si prova ad aprire un file non c'è garanzia che l'operazione vada a buon fine. Proseguire nel codice del programma in caso in cui l'operazione fallisca, porterebbe a comportamenti indeterminati, che potrebbero portare alla chiusura anticipata del programma o peggio.

Risulta quindi necessario controllare ogni apertura di un file, come mostrato nel listato 8.5:

Listing 8.5: Controllo sull'apertura di un file

```
1     if (!in) {
2         std::cout << "Errore nell'apertura del file input.
      txt" << std::endl;
3         return 1;
4     }
5     if (!out) {
6         std::cout << "Errore nell'apertura del file output.
      txt" << std::endl;
7         return 1;
8     }
9 }
```

Alla riga 1 e alla 5 viene mostrata la scrittura idiomatica per verificare se un file è stato aperto correttamente, in cui, nella condizione

dell'`if`, viene indicato il nome della variabile che rappresenta lo *stream* preceduto dall'operatore di negazione, che potrebbe essere letto come «se il file `in` **non** è stato aperto correttamente, allora gestisci il problema».

Nel caso di apertura fallita, verrà eseguito il corpo dell'`if` e si gestirà il problema.

Mentre la verifica sulla correttezza o meno dell'apertura è un'operazione che viene eseguita sempre allo stesso modo, la gestione del problema è fortemente dipendente dal contesto. Nell'esempio 8.5 viene mostrato un messaggio d'errore e viene chiuso il programma[12]: questo modo di procedere è indicato quando si ritiene che la non apertura del file sia un errore irrimediabile e che il programma non possa proseguire se il file non è stato aperto correttamente. Altri modi di procedere potrebbero essere:

- dopo aver verificato il fallimento dell'apertura, si richiede all'utente di inserire scegliere un file differente, che, in un'applicazione testuale, vorrebbe dire di inserire un nuovo nome del file, eventualmente con un percorso differente
- anche se il file non è stato aperto correttamente, si evita di procedere con eventuali letture/scritture, ma il programma continua nella sua esecuzione, eventualmente segnalando che si è verificato un problema

12: Si può notare che viene ritornato il valore 1 anziché 0: l'idea è che il valore di ritorno della funzione `main` vale 0 quando tutto è andato correttamente, qualsiasi altro valore indica invece che qualcosa è andato storto.

A questo punto ci si potrebbe domandare quali siano le condizioni che portano al fallimento nell'apertura di un file. Per quanto riguarda un file in lettura, i problemi potrebbero essere:

- il file non esiste: generalmente, in un programma didattico, questo vuol dire che si è scritto erroneamente il nome del file oppure non si trova nel percorso che ci si era immaginati fosse
- il file non può essere letto: in questo caso il file esiste, ma per un discorso di proprietà e permessi, non è accessibile al proprio programma

Per quanto riguarda invece l'apertura di un file in scrittura le possibilità sono un po' diverse:

- il file non può essere scritto: il file esiste, ma per un discorso di proprietà e permessi, non è accessibile al proprio programma
- il file non è scrivibile: si pensi ad esempio a file in sola lettura memorizzati all'interno di supporti non scrivibili, come ad esempio i CD

Per quanto riguarda la scrittura, si sarà notato che il fatto che il file non esista non è di per sé un problema, in quanto, ogni volta che si proverà ad aprire un file in scrittura, questo verrà creato *ex-novo*, eventualmente sovrascrivendo un file con lo stesso nome che

esiste già[13]. Anche se questo comportamento potrebbe sembrare «pericoloso», per l'utilizzo che si farà dei file, è del tutto sensato e ragionevole.

13: Come già accennato, sarebbe possibile anche aprire un file in modalità *append*, in cui i dati precedentemente presenti verrebbero preservati, ma non è una modalità che verrà trattata.

Lettura/scrittura su file

Una volta che il file è stato aperto correttamente, si possono effettuare le operazioni di lettura/scrittura desiderate. A questo punto il meccanismo di gestione dei file del C++ si dimostra elegante, poiché permette di utilizzare le stesse modalità già viste finora per i flussi `std::cin` e `std::cout`, semplicemente utilizzando le variabili di tipo `fstream` create nel modo opportuno.

Listing 8.6: Lettura di numeri all'interno di un file

```
    int n;
    std::vector<float> v;
    in >> n;
    for (int i = 0; i < n; ++i) {
        float temp;
        in >> temp;
        v.push_back(temp);
    }
}
```

Nell'esempio del listato 8.6 si suppone di voler leggere un file la cui organizzazione interna sia fatta in modo da aver un intero N come primo elemento e, successivamente, una serie di N numeri `float`. Pertanto, si procederà inizialmente alla lettura dell'intero, che verrà memorizzato nella variabile `n` e poi, utilizzando un ciclo composto da `n` iterazioni, si leggeranno gli `n` numeri `float`, che verranno memorizzati in un vettore.

In situazioni in cui è nota a priori la struttura interna del file, il modo di procedere sarà analogo, adattandolo opportunamente alla specifica struttura descritta nel testo del problema o alle scelte fatte dal programmatore.

Nelle situazioni in cui invece il file può contenere qualsiasi testo senza una struttura specifica, l'idea generale sarà quella di leggere il contenuto linea per linea all'interno di una stringa, per poi elaborare le stringhe in maniera consona alle richieste del problema.

Nel listato 8.7 si può vedere che il modo idiomatico di procedere è quello di utilizzare la funzione `getline`, già vista in precedenza nel capitolo sulle stringhe, come condizione per il ciclo `while`, che rappresenta l'idea «leggi una stringa alla volta fino a quanto non è finito il file». La `getline` si comporta come già visto per le stringhe, legge un'intera linea di testo fino a quando non incontra un «a capo» e la copia nella stringa passata come secondo parametro, in questo esempio `linea`. All'interno del ciclo si potrà fare quello che si desidera sulla stringa appena letta, dove in questo caso si

è deciso di inserirla all'interno del vettore `linee`. Al termine del ciclo il vettore `linee` conterrà tutte le righe del file, anche eventuali linee *vuote* che dovessero essere presenti.

Listing 8.7: Lettura *generica* di tutte le righe di un file di testo

```
    std::string linea;
    std::vector<std::string> linee;
    while(getline(in, linea)) {
        linee.push_back(linea);
    }
```

Il processo di scrittura è ancora più semplice, nel senso che la struttura di quanto scritto viene decisa dal programmatore, allo stesso modo in cui si procede per scrivere sul terminale, solo che al posto di usare `std::cout` si utilizzerà la variabile (o le variabili) creata per la gestione del file in scrittura.

Listing 8.8: Scrittura su file.

```
    out << "Nel vettore sono presenti " <<
        v.size() << " numeri." << std::endl;
    for (int i = 0; i < v.size(); ++i) {
        out << i << " -> " << v.at(i) << std::endl;
    }
```

Utilizzando il codice presente nel listato 8.8 per scrivere in un file gli elementi presenti in un vettore `v`, verrà prima scritta una riga contenente il numero di elementi del vettore e, successivamente, la lista degli elementi, ognuno preceduto dalla sua posizione nel vettore, producendo un file al cui interno ci sarà del testo simile a questo:

```
Nel vettore sono presenti 3 numeri.
0 -> 4.5
1 -> 6.7
2 -> 9.3
```

Chiusura di un file

Dopo aver fatto le operazioni di interesse per un programma, viene richiesto di «chiudere» il file. L'operazione di chiusura è un messaggio che viene mandato al sistema operativo per dire di completare eventuali operazioni lasciate in sospeso sul file e successivamente di renderlo disponibile per altri programmi che volessero manipolarlo.

Per quanto riguarda il primo aspetto, la chiusura di operazioni lasciate in sospeso, è importante sapere che in generale le operazioni di I/O sono *bufferizzate*, che significa, senza entrare troppo nei dettagli, che il momento in cui viene eseguita una lettura/scrittura non coincide necessariamente con il momento in cui l'operazione viene fisicamente effettuata[14] e, anzi, soprattutto per le scritture,

14: Il motivo risiede nel fatto che, essendo le operazioni effettuate dalla CPU migliaia di volte più rapide delle singole letture/scritture sui dispositivi di I/O, se il processore dovesse aspettare il completamento di un'operazione di I/O per procedere all'istruzione successiva, si creerebbero degli enormi problemi di performance.

generalmente in un momento che potrebbe anche essere molto in là nel tempo (sempre dal punto di vista del processore, che opera a frequenze del GHz). La chiusura forza il completamento di tutte le operazioni sospese, evitando che il file venga liberato alla chiusura del programma con operazioni non terminate, con il rischio di corrompere i dati contenuti nel file[15].

15: Nella pratica questo può succedere, ad esempio, con file di grandi dimensioni in cui vengono effettuate molte operazioni di scrittura prima della chiusura del programma.

Relativamente invece alla possibilità di renderlo disponibile ad altri programmi, anche in questo caso senza approfondire troppo, è importante sapere che se un file viene aperto da un programma, il S.O. lo rende inaccessibile a qualsiasi altro programma che cerchi di aprirlo o, eventualmente, lo potrebbe rendere accessibile in sola lettura, perché chiaramente l'accesso condiviso alla stessa risorsa, senza un qualche tipo di controllo di condivisione, potrebbe portare a situazioni a inconsistenze nei dati. In questo caso la chiusura permette al S.O. di *liberare* il file e renderlo nuovamente accessibile a tutti gli altri programmi.

Per chiudere un file è sufficiente utilizzare il metodo `close`, come si può vedere nel listato 8.9, dove vengono chiusi i file che erano stati aperti nel listato 8.4.

Listing 8.9: Chiusura di file.

```
1
2     in.close();
3     out.close();
```

Dopo aver effettuato la chiusura, le variabili `in` e `out` non saranno più associate a nessun file, quindi un loro eventuale utilizzo successivo porterà a una situazione indeterminata, e quindi erronea.

La chiusura dei file deve avvenire quando si ritiene che non sia più necessario utilizzarli: in molti programmi didattici questo viene fatto prima del `return 0` che si trova in fondo al `main`, ma, in generale, il momento migliore per chiuderli è appena il programma non ha più bisogno di usarli. Questo minimizza le possibilità di corruzione del contenuto nella malaugurata eventualità di un *crash* del programma, poichè, se un file fosse ancora aperto quando il programma che lo sta gestendo verrà chiuso inaspettatamente in seguito a un errore, ad esempio a causa dell'accesso a un vettore fuori dal suo range o a seguito di una divisione per zero, ne potrebbe risultare una corruzione dei dati.

8.3. Come rendere persistenti le informazioni

Finora si è visto come poter leggere e scrivere informazioni all'interno di file: in questo paragrafo verrà spiegato un modo

un'organizzazione del codice molto semplice, ma di utilizzo generale, che permetterà di rendere persistenti le informazioni trattate all'interno di un programma.

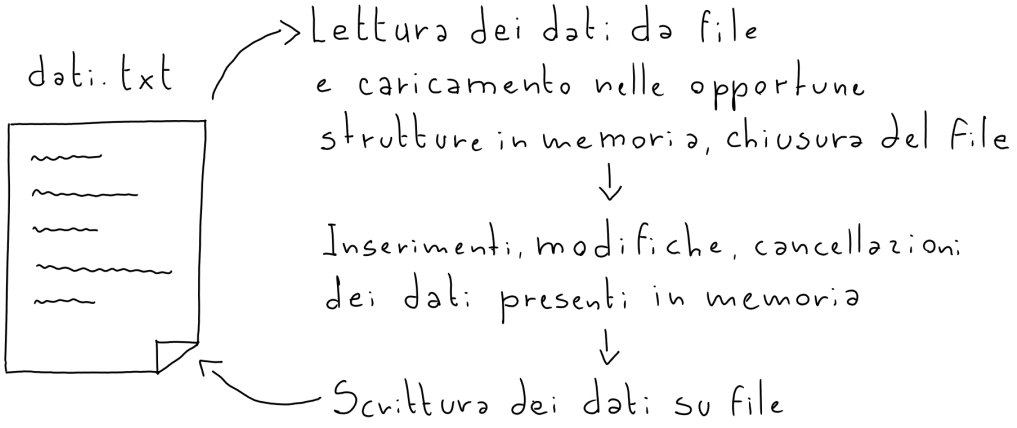


Figura 8.3.: Come rendere persistenti le informazioni in un programma

Questo modello generale è esposto nella Figura 8.3, dove si può vedere che esistono tre momenti distinti:

1. viene letto un file, chiamato nell'esempio `dati.txt`, che contiene i dati memorizzati in precedenti esecuzioni del programma. Questi dati, organizzati opportunamente, andranno a finire all'interno di variabili che potranno essere di tipo primitivo, ma anche vettori, strutture o combinazioni di questi, per essere utilizzati nel programma. Appena finita la lettura questo file verrà chiuso, in quanto non più utile al resto del programma
2. il programma, a questo punto, avrà a disposizione i dati per poter fare le operazioni per le quali è stato progettato: potrà quindi inserire nuovi dati, modificare dati già presenti e/o cancellarli, tutto lavorando sulle variabili come visto nei capitoli precedenti
3. al momento della terminazione del programma, per scelta dell'utente o perché sono state effettuate tutte le operazioni previste, i dati presenti in memoria verranno *riversati* nuovamente nel file `dati.txt`, mantenendo la stessa organizzazione definita in precedenza, in modo che a una successiva esecuzione del programma queste fasi possano essere ripetute nello stesso modo.

Come si dovrebbe intuire, questo approccio ha il vantaggio di poter riutilizzare programmi scritti in precedenza, aggiungendo solo la parte 1 e la parte 3, e utilizzando i programmi già scritti come parte 2, avendo cura di riempire le loro variabili con i dati presenti nel file e di salvarle successivamente sempre nello stesso file[16].

Nel caso che al punto 1 la lettura del file fallisse, in generale non dovrebbe essere considerato un problema, perché potrebbe sem-

16: In questo esempio modello viene utilizzato un file solo, ma nulla vieta, per situazioni complesse, di utilizzarne di più, sempre con l'accortezza di fare in modo che ciò che verrà scritto sia nella forma giusta per essere letto in esecuzioni successive.

plicemente essere la prima esecuzione del programma, in cui non sono evidentemente presenti dati salvati da un'esecuzione precedente. Il programma procederà normalmente, poiché le variabili saranno *vuote* come lo sarebbero se non venissero utilizzati file. Quando poi il programma terminerà e si arriverà quindi al punto 3, verrà creato il file e ogni esecuzione successiva lo troverà e avrà la possibilità di leggerlo.

8.4. Esempio: lettura di un file CSV

Uno degli aspetti che non sono stati presi in considerazione nel paragrafo precedente è come organizzare i dati in un file perché possano essere letti e scritti nella maniera più comoda ed efficace possibile.

Non esiste una ricetta generale, dipende molto dalla complessità dei dati che si vogliono memorizzare e, nel tempo, sono stati proposti molti formati testuali diversi, come YAML, XML, JSON e altri, ognuno con le proprie caratteristiche. In questo esempio si utilizzerà uno dei formati testuali più semplici e «antichi», il formato CSV. Come dice l'acronimo, **C**omma **S**eparated **V**alues, l'idea fondamentale è di associare a ogni riga del file una serie di valori, uno per ognuno dei dati di interesse, e separare ogni valore da quello successivo tramite un segno, normalmente di interpunzione, come la virgola (da cui *comma*), ma potrebbero essere anche i due punti, il punto e virgola o altro.

Per mostrare un possibile approccio, si utilizzeranno come dati quelli definiti nel Capitolo 7 (Strutture in C++), come formalizzati nel listato 7.8 alla pagina 162. Questo risulta già essere un esempio complesso, ma così sarà possibile evidenziare come affrontare situazioni molto generali.

Come prima cosa è necessario scegliere che carattere utilizzare come separatore di campi, e in questo esempio si utilizzerà il punto e virgola, poiché difficilmente comparirà nei dati che riguardano gli studenti e le valutazioni[17].

17: La presenza del carattere di separazione all'interno dei dati complica ovviamente la possibilità di interpretare in maniera univoca quanto salvato nei file: si può gestire questa problematica, ma non verrà fatto qua.

Partendo dalla `Data`, che è formata da tre interi, la scelta più ovvia è quella di scrivere i tre interi nell'ordine presente nella struttura, ottenendo la forma `giorno;mese;anno`. Anche per quanto riguarda la valutazione non ci sono problemi particolari, in quanto ogni campo può essere rappresentano separandolo dagli altri con il `;`, mentre la scelta per lo studente è un attimo più complessa, in quanto contiene un vettore di strutture di tipo `Valutazione`, che può avere una lunghezza qualsiasi. La scelta adottata sarà quella di indicare il numero N di valutazioni dello studente come ultimo

campo dopo la `sezione` e, nelle successive N righe, scrivere in ognuna la singola valutazione.

L'ultimo problema da risolvere è come memorizzare più studenti: siccome non è pratico scrivere un file per ogni studente, la scelta più semplice è di scrivere un file che avrà alla prima riga il numero di studenti contenuti e, successivamente, conterrà su varie righe i dati di ogni studente, tanti quanti indicati alla prima riga.

Cercando di rappresentare in uno schema generale le scelte fatte, si otterrà quando indicato nel seguito:

Listing 8.10: Formato CSV per la memorizzazione degli studenti.

```
numero_studenti
nome;cognome;giorno;mese;anno;classe;sezione;n_valutazioni
voto;materia;giorno;mese;anno
...
```

dove, per meglio comprendere questa rappresentazione, nel seguente esempio vengono indicati dei possibili valori:

```
2
Alessandro;Bilungi;19;6;2008;3;A;2
7.5;Informatica;11;1;2024
6.5;Italiano;23;2;2024
Cristina;Troffoli;22;6;2008;3;A;3
8.5;Informatica;11;1;2024
8.5;Italiano;1;3;2024
9;Sistemi e reti;4;4;2024
```

Una volta fatte le scelte che si ritengono più opportune per la rappresentazione dei dati, bisogna procedere con la creazione delle funzioni per la lettura e il salvataggio delle informazioni presenti nel file, in modo poi da mettere in pratica il modello presentato in Figura 8.3. Prima ancora di procedere, è necessario domandarsi come risolvere il problema di spezzare le singole linee del file CSV in modo da ottenere i singoli *token*, cioè i dati compresi tra un punto e virgola e il successivo. Il C++, a differenza di altri programmi, non ha allo stato attuale un metodo di libreria che risolva questo problema, quindi viene proposto questo il codice in 8.11, che dovrebbe essere sufficientemente robusto per poter gestire tutte le situazioni «normali».

Listing 8.11: Funzione per lo *splitting* di una riga indicandone il separatore.

```
1   std::vector<std::string> split(const std::string& s,
    char delimiter = ' ')
2   {
3       std::stringstream in(s);
4       std::string temp;
5       std::vector<std::string> v;
6       while(getline(in, temp, delimiter)){
7               v.push_back(temp);
```

```
        }
        return v;
    }
```

Senza entrare nel dettaglio della funzione, il nome `split` è stato scelto poiché è quello più comunemente utilizzato per funzioni che risolvono questo problema. I due parametri indicano rispettivamente la stringa oggetto dell'operazione di *splitting* e il carattere che verrà utilizzato come separatore[18]. Come valore di ritorno è stato scelto un vettore di stringhe, che conterrà come elementi ogni singolo *token* prodotto dalla funzione. Se ad esempio la stringa fosse:

18: La sintassi con l'uguale presente nel secondo parametro indica che, se durante la chiamata venisse indicato un solo parametro, al secondo verrebbe automaticamente assegnato il carattere spazio, che quindi è stato scelto come valore di default.

```
Alessandro;Bilungi;19;6;2008
```

il vettore ritornato conterrà al suo interno gli elementi

```
{"Alessandro","Bilungi","19","6","2008"}
```

Utilizzando la funzione `split`, il listato 8.12 mostra una possibile implementazione della lettura degli studenti presenti in un file, formattato[19] secondo le definizioni date in precedenza.

19: In tutti gli esercizi del testo si darà per assunto che il file sarà organizzato esattamente come prevede il formato scelto dal programmatore: è chiaro che anche una piccola discordanza potrebbe portare al fallimento della lettura.

Listing 8.12: Funzione per il caricamento di un vettore di studenti.

```
std::vector<Studente> carica_studenti(const std::string
 &nomefile) {
std::vector<Studente> risultato;
std::ifstream in(nomefile);
if (in) {
    int n_studenti;
    std::string riga;
    getline(in, riga);
    n_studenti = std::stoi(riga);
    for (int i = 0; i < n_studenti; ++i) {
        Studente s;
        getline(in, riga);
        std::vector<std::string> tokens =
            split(riga, ';');
        s.nome = tokens.at(0);
        s.cognome = tokens.at(1);
        s.nascita.giorno = std::stoi(tokens.at(2));
        s.nascita.mese = std::stoi(tokens.at(3));
        s.nascita.anno = std::stoi(tokens.at(4));
        s.classe = std::stoi(tokens.at(5));
        s.sezione = tokens.at(6).at(0);
        int n_valutazioni = std::stoi(tokens.at(7));
        for (int i = 0; i < n_valutazioni; ++i) {
            Valutazione v;
            getline(in, riga);
            std::vector<std::string> tokens =
                split(riga, ';');
            v.voto = std::stof(tokens.at(0));
            v.materia = tokens.at(1);
            v.data.giorno = std::stoi(tokens.at(2));
```

```
30                  v.data.mese = std::stoi(tokens.at(3));
31                  v.data.anno = std::stoi(tokens.at(4));
32                  s.valutazioni.push_back(v);
33              }
34              risultato.push_back(s);
35          }
36      }
37      return risultato;
38  }
```

Come si può vedere, la funzione `carica_studenti` riceve come unico parametro il nome del file che contiene i dati degli studenti, organizzato come indicato in precedenza, e ritorna il vettore con all'interno tutti i dati degli studenti e delle loro valutazioni.

Alla riga 3 avviene il tentativo di apertura del file: in questo caso il fallimento dell'apertura non comporta nessun problema particolare, in quanto la funzione ritornerà comunque il vettore degli studenti, solo che sarà vuoto, perché non verranno effettuate le operazioni comprese all'interno dell'`if` alla riga 4.

Se il file viene aperto correttamente, viene letta la prima riga, quella che contiene il numero degli studenti, e il valore viene convertito in un intero[20] e assegnato alla variabile `n_studenti`.

20: Si ricorda infatti che tutte le letture della funzione `getline` producono stringhe, che vanno eventualmente convertite nel tipo corretto.

Segue quindi un ciclo, con un'iterazione per ogni studente, dopo viene letta la riga con le sue informazioni, il suo contenuto viene splittato all'interno del vettore `tokens` e, nelle linee di codice dalla 14 alla 20, i singoli campi vengono assegnati ai corrispondenti campi della struttura `Studente` indicata con `s`, effettuando eventuali conversioni come nel caso della data e della classe.

L'ultimo *token* contiene il numero di valutazioni dello studente corrente e viene quindi utilizzato per indicare, nel successivo ciclo `for`, il numero di iterazioni da effettuare, una per ogni valutazione. A questo punto viene ripreso lo stesso schema utilizzato per lo studente, leggendo una linea del file CSV, splittandola e inserendo i valori all'interno di una valutazione, che infine verrà inserita (linea 32) nel vettore delle valutazioni dell'utente corrente.

Dopo aver letto e inserito le valutazioni, lo studente, adesso completo di tutti i suoi dati, verrà inserito nel vettore degli studenti (linea 34), che verrà infine ritornato al chiamante.

Per completare l'esempio, per la parte di scrittura (o salvataggio) dei dati, viene proposta la funzione `salva_studenti` presente nel listato 8.13.

Listing 8.13: Funzione per il salvataggio su file di un vettore di studenti.

```
1   bool salva_studenti(const std::vector<Studente>
2       &studenti, const std::string &nomefile) {
3       std::ofstream out(nomefile);
4       if (!out) {
5           return false;
6       }
7       out << studenti.size() << std::endl;
8       for (int i = 0; i < studenti.size(); ++i) {
9           Studente s = studenti.at(i);
10          Data d = s.nascita;
11          int n_valutazioni = s.valutazioni.size();
12          out << s.nome << ";" << s.cognome << ";"
13          << d.giorno << ";" << d.mese << ";"
14          << d.anno << ";"
15          << s.classe << ";" << s.sezione << ";"
16          << n_valutazioni << std::endl;
17          for (int i = 0; i < n_valutazioni; ++i) {
18              Valutazione v = s.valutazioni.at(i);
19              out << v.voto << ";" << v.materia << ";"
20              << v.data.giorno << ";"
21              << v.data.mese << ";"
22              << v.data.anno << std::endl;
23          }
24      }
25      return true;
26  }
```

In questo caso, la funzione riceverà come unico parametro il vettore degli studenti che dovranno essere salvati su file e restituirà un valore *booleano* per indicare se il salvataggio è andato a buon fine (`true`) oppure no (`false`).

Dopo l'apertura del file e il relativo controllo, sarà sufficiente scrivere nella prima riga del file il numero di elementi presenti nel vettore e successivamente iterare su tutto il vettore, scrivendo ogni campo dello studente attuale all'interno del file, usando il punto e virgola come separatore.

Per rendere il codice un po' più leggibile, lo studente attuale è stato copiato nella variabile `s` (linea 9) e il numero di valutazioni nella variabile `n_valutazioni` (linea 10). Quest'ultima variabile verrà usata poi per stampare le valutazioni dello studente, utilizzando un altro ciclo `for` e, anche qua, usando la variabile *d'appoggio* `v` per rendere il successivo codice meno prolisso.

Se tutto è stato scritto correttamente, questa funzione dovrà produrre un file con un formato che rispetti perfettamente le specifiche presenti nella definizione 8.10, in modo che la funzione `carica_studenti` possa leggerlo e quindi sia possibile implementare l'architettura mostrata in Figura 8.3.

8.5. Esercizi

Studi

1. Creare un file nominato «numeri.txt», contenente 10 numeri interi. Successivamente scrivere un programma che li legga tutti e 10 e stampi la loro somma e il loro prodotto.
2. Creare un file nominato «temperature.txt», contenente 10 temperature espressi in gradi centigradi con una cifra decimale. Successivamente scrivere un programma che le legga tutte e 10 e stampi la media.
3. Scrivere un programma che, chiesto all'utente un numero compreso tra 1 e 10, stampa in un file la tabellina di quel numero.
4. Scrivere un programma che, chiesto all'utente un numero intero positivo `N` e un carattere `c`, stampi un quadrato pieno formato tutto da caratteri `c` con lato di lunghezza `N`.
5. Scrivere un programma che, chiesto all'utente un numero intero positivo `N` e due caratteri `c` e `d`, stampi un quadrato pieno formato tutto dai caratteri `c` e `d`, che si alternano come in una scacchiera, con lato di lunghezza `N`.
6. Creare un file nominato «numeri.txt», che contenga un numero qualsiasi di numeri interi. Scrivere poi un programma che stampi il massimo e il minimo fra i numeri contenuti nel file.
7. Creare un file nominato «ricerca.txt», che contenga un numero qualsiasi di numeri interi positivi. Scrivere poi un programma che chieda all'utente di inserire un numero `N` e comunichi poi se si trova o meno nel file. Il programma deve continuare con la ricerca fino a quando l'utente non inserisce -1.
8. Creare un file nominato «insieme.txt», contenente un intero positivo `N` e successivamente `N` numeri interi. Scrivere poi un programma che legga gli N numeri interi. Trovati i valori `minimo` e `massimo` di questo insieme di numeri, stampi in un secondo file solo i numeri `P` che soddifino *almeno* una delle seguenti condizioni:

$$P - \text{minimo} < 10$$

$$\text{massimo} - P < 10$$

9. Creare un file nominato «decimali.txt», contenente un intero positivo `N` e successivamente `N` numeri con la virgola. Scrivere poi un programma che stampi in un secondo file solo i numeri che hanno una sola cifra decimale.
10. Creare un file nominato «nomi.txt», contenente un nome per riga. Successivamente scrivere un programma che li legga tutti e stampi una riga per ogni nome, con la scritta «Ciao *nome*», dove al posto di *nome* ci devono essere i nomi letti dal file.
11. Creare un file nominato «frasi.txt», contenente una frase per ogni riga. Successivamente scrivere un programma che le legga tutte e le riscriva in un secondo file dall'ultima alla prima.
12. Creare un file nominato «frasi.txt», contenente una frase per ogni riga. Successivamente scrivere un programma che le legga tutte e le riscriva in un secondo file in modo che ogni frase sia riscritta al contrario. Ad esempio «Ciao a tutti» diventerebbe «ittut a oaiC».
13. Creare un file nominato «frasi.txt», contenente una frase per ogni riga. Successivamente scrivere un programma che le legga tutte e le riscriva in un secondo file in modo che ogni frase sia riscritta con le parole in ordine invertito. Ad esempio «Ciao a tutti» diventerebbe «tutti a Ciao». Si usi lo spazio come carattere separatore di parole.

Esercizi

1. Scrivere un programma che legga da un file una serie di N numeri interi in base 10 e li riscriva in un altro file convertiti in base 16. Il valore N è il primo che si trova nel file, seguono poi gli N numeri.
2. Scrivere un programma che stampi in un file la tabella ASCII di tutti i caratteri a partire dal carattere 33, il punto esclamativo (!), fino al carattere 126, la tilde (~). Eventualmente formattare la stampa in modo che il file sia comodo da leggere e siano evidenti le relazioni tra un codice e il carattere che rappresenta.
3. Scrivere un programma che permetta di cifrare un file contenente un testo qualsiasi, salvando il testo cifrato in un altro file. Per far questo, il programma chieda all'utente il nome del file da cifrare euna chiave di cifratura (si veda a tal proposito l'esercizio Esercizio 7) e, utilizzando tale chiave, cifri il testo contenuto nel file indicato, salvandolo in un file che dovrà essere scelto dall'utente inserendone il nome. Una volta implementata correttamente questa parte, si proceda a fare anche la funzionalità per decifrare un file, data la chiave.
   Per tutto quanto non indicato esplicitamente nel testo, si proceda nel modo che si ritiene più adatto, specificando eventualmente alcuni aspetti nei commenti o come indicazioni per l'utente.
4. Scrivere un programma che legga un file in formato CSV in cui vengono memorizzati i dati di una corsa utilizzando tre campi, che rappresentano la distanza in metri percorsa dall'inizio della corsa, il tempo dall'inizio della corsa e la frequenza cardiaca in quell'istante. Ogni riga rappresenta la lettura ottenuta da un orologio contenente un GPS, che viene effettuata ogni 5 secondi. Un esempio, con solo i primi 20 secondi di corsa potrebbe essere questo:

   ```
   17;5;70
   32;10;82
   49;15;94
   65;20;94
   ```

   Dopo averlo letto, dovrà scrivere in un altro file delle informazioni di sintesi della corsa così organizzate:

   a) per ogni chilometro percorso, verrà indicato la velocità media, sia in chilometri all'ora che in minuti-secondi impiegati per percorrere un chilometro (ad esempio, 12 Km/h equivalgono a 5 minuti al chilometro), e la frequenza cardiaca media
   b) per l'ultimo tratto, che potrebbe essere inferiore a un chilometro, oltre a quanto indicato per le righe precedenti, anche la lunghezza percorsa
   c) il chilometro più veloce e quello più lento

   L'esatta organizzazione delle informazioni di questo file è lasciata libera.

Progetti

Stampa unione

La *stampa unione* è un processo, presente in programmi come Microsoft Word o LibreOffice, per creare una serie di documenti personalizzati, che, partendo da uno schema o *template* a da una serie di N dati, produca N documenti, ognuno uguale all'altro, però personalizzati utilizzando gli N dati.

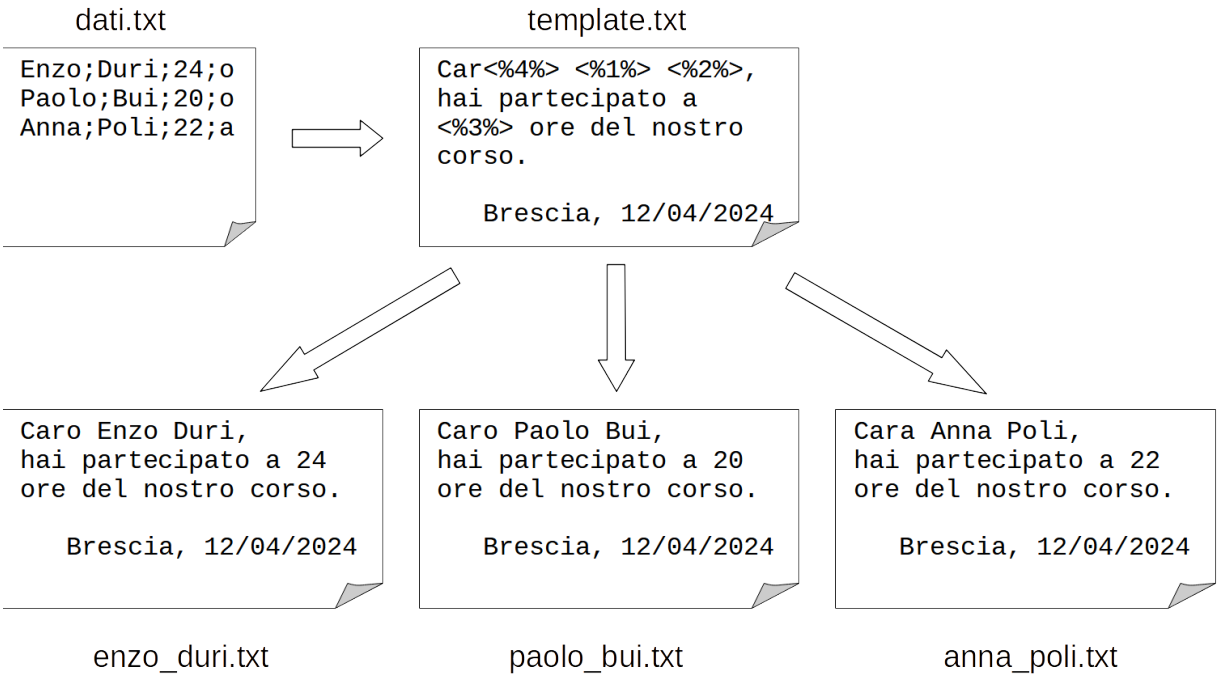


Figura 8.4.: Esempio di stampa unione con file separati.

Si supponga di avere, ad esempio, i dati di N studenti di una classe, ognuno identificato dal proprio nome e cognome e, per ognuno, il numero di ore a cui ha partecipato al corso extra-curricolare «Programmazione in Ada». Volendo produrre un attestato di partecipazione per ogni studente, al posto di scriverli uno per uno, la procedura di stampa unione permette di creare un modello, che in alcune parti conterrà dei segnaposti per nome, cognome e numero di ore, e creerà automaticamente N documenti personalizzati, uno per ogni studente.

Il programma, partendo da due file, uno con una serie di dati serializzati utilizzando CSV e uno con un modello (*template*) strutturato come definito successivamente, crei un terzo file contenente l'unione dei primi due, come indicato in precedenza. Per il *template*, si inseriscano dei *tag* della forma `<%i%>`, dove al posto di *i* è presente un numero che indica la posizione, all'interno di ogni riga del file CSV, del dato che dovrà essere sostituito al *tag* nel processo di unione.

All'utente verrà chiesto di inserire il nome del file contenente i dati, il nome del file del *template* e se desidera che venga stampato un solo file oppure un file per ognuna delle righe presenti nel file di dati: in quest'ultimo caso, l'utente dovrà fornire anche una stringa che indica come deve essere composto il file: in Figura 8.4, il file dei dati si chiama `dati.txt`, il file di *template* si chiama `template.txt` e l'utente ha richiesto la scrittura di più file, fornendo come stringa `<%1%>_<%2%>.txt`, che quindi produrrà i 3 file indicati in figura.

Registro elettronico personale

Lo scopo di questo programma è di implementare un piccolo registro personale per memorizzare i voti presi durante un anno scolastico. Per ogni materia si possono avere più voti, e ogni voto è caratterizzato dal punteggio (6, 7.5, 4.75) che è un valore reale, dalla data in cui è stato preso e dalla tipologia (scritto, orale o altro).

Come prima cosa viene chiesto di definire le strutture opportune da utilizzare in questo programma.

Alla prima esecuzione del programma, il registro è vuoto, quindi quando si vuole inserire un nuovo voto ci sono due possibilità:

- la materia è già presente e quindi viene inserito il voto, con le sue caratteristiche
- la materia non è presente, e quindi prima verrà inserita quella e successivamente il voto, con le sue caratteristiche

Oltre alla possibilità di inserire nuovi voti il programma dovrà anche:

- mostrare l'elenco delle materie la cui media dei voti è insufficiente
- mostrare tutti i voti di una materia scelta dall'utente
- mostrare tutti i voti di tutte le materie e la media totale
- qualsiasi altra funzionalità che si ritenga necessaria

Per quanto non indicato esplicitamente nel testo, si rimanda allo studente di fare delle scelte ragionevoli che lo rendano un registro personale utilizzabile.

Il programma dovrà inoltre caricare e salvare tutti i voti su un file, in modo che la situazione sia preservata alla chiusura del programma. Anche il formato dei dati all'interno del file viene lasciato libero, si consiglia ovviamente di seguire quanto visto nel paragrafo d'esempio sulla lettura/scrittura di dati su file.

Nuovo Cinema Paradiso

Vi hanno commissionato la creazione di un sistema per la gestione delle prenotazioni in un cinema.

Forma base

Il sistema deve essere usato da un dipendente del cinema che riceve telefonate di persone che vogliono prenotare un posto a un cinema di forma rettangolare, con N file e M colonne. Il sistema deve permettere al dipendente di fare queste due semplici operazioni:

- inserisci prenotazione: il dipendente inserisce una nuova prenotazione con il nome e cognome della persona che prenota e il numero di riga e colonna del posto. Se il posto è già prenotato inserisce un nuovo posto e così via finché non trova un posto libero
- mostra prenotazioni: il sistema mostra la sala a schermo, in un modo accettabile essendo un programma a consolle, con l'indicazione dei posti liberi e dei posti prenotati

Il programma deve essere in grado di salvare le prenotazioni in modo da ritrovare le prenotazioni precedenti quando si riavvia.

Idee per il miglioramento

- proponi prenotazione: il sistema, nel momento in cui viene fatta una prenotazione, propone un posto libero secondo un algoritmo ragionevole (ad esempio scegliendo il primo posto libero vicino al centro della sala)
- prenotazioni multiple: se un cliente vuole prenotare più posti, non vengono inserite le prenotazioni una ad una, ma si può inserire un intervallo di posti

- elimina prenotazione: se un cliente vuole eliminare una prenotazione indica il proprio nome e cognome oppure il codice della prenotazione (se si vuole usare un codice deve essere già previsto nella fase di prenotazione e deve essere memorizzato insieme agli altri dati)
- gestione di più spettacoli: il sistema prevede che la prenotazione non riguardi un unico spettacolo, ma molteplici. Il sistema deve quindi avere una parte che deve permettere di inserire nuovi spettacoli (si può tralasciare la modifica e la cancellazione) e al momento della prenotazione deve essere chiesto anche quale spettacolo si vuole prenotare. A questo punto bisogna memorizzare su file anche le informazioni dei vari spettacoli e le prenotazioni devono fare riferimento ad essi.
- stampa del biglietto: il sistema deve prevedere la stampa dei biglietti delle prenotazioni. Per farla facile facile si può stampare il singolo biglietto su file, con le informazioni e gli abbellimenti che si vogliono, dandogli un nome che lo identifichi e successivamente l'operatore lo stamperà come file.
- versione grafica di tutto o parte di quello proposto: una volta realizzato il sistema, se ben strutturato, si può pensare di trasformarlo in un'applicazione grafica, con tutti gli abbellimenti e adattamenti del caso, usando la libreria grafica Raylib.

Appendice

Operatori matematici, relazionali e logici, funzioni

A.

Nella maggior parte degli algoritmi utilizzati in questo testo è necessario produrre dei risultati attraverso la combinazione di operatori matematici, relazionali e logici. Questi operatori rappresentano l'insieme di operazioni che un computer è in grado di svolgere grazie al proprio *hardware* e quindi possono essere visti come i *mattoncini* fondamentali con cui costruire ogni programma, da quelli didattici affrontati in questo testo a quelli più avanzati che si trovano nel mondo reale[1].

Le funzioni sono invece degli strumenti che ogni linguaggio mette a disposizione per ampliare l'insieme delle operazioni che è in grado di svolgere un computer.

1: Nei computer reali sono state nel tempo introdotte altre operazioni via via più complesse che migliorano le capacità di calcolo e l'efficienza di elaborazione, ma questo non toglie che gli operatori che verranno visti in questo capitolo sono quelli strettamente indispensabili per ottenere qualsiasi risultato.

A.1. Operatori matematici

Gli operatori *matematici* sono quelli già noti, in particolare sono quelli che rappresentano le quattro operazioni fondamentali (addizione, sottrazione, moltiplicazione e divisione) a cui verrà aggiunto l'operatore di *resto della divisione intera*, chiamato normalmente *modulo*.

Tabella A.1.: Operatori matematici

Operazione	Simbolo
Addizione	`+`
Sottrazione	`-`
Moltiplicazione	`*`
Divisione	`\`
Modulo o resto	`%`

I risultati delle operazioni che coinvolgono questi operatori sono quelli usuali della matematica, ma va tenuto presente che nel calcolatore esiste una distinzione netta tra i numeri *interi* e i numeri *reali*, derivante dalla struttura dell'hardware del computer: la conseguenza più visibile è che un'operazione tra due numeri interi darà come risultato sempre un numero intero, quella tra due numeri reali sempre un numero reale, quella tra un intero e un reale darà come risultato un numero reale.

Va infine ricordato che se si vogliono utilizzare più operatori nella stessa espressione il calcolo viene effettuato rispettando le solite precedenze già note in matematica, dove moltiplicazione, divisione e modulo hanno precedenza maggiore di addizione e sottrazione. Volendo cambiare la precedenza vanno utilizzate le parentesi *tonde*

in numero arbitrario, stando solo attenti che ad ogni parentesi aperta ne corrisponda una chiusa.

Nella Tabella A.2 si possono vedere degli esempi di utilizzo che chiariscono il comportamento da questi operatori.

Tabella A.2.: Esempio di utilizzo degli operatori matematici

Operazione	Risultato	Note
`1 + 1`	`2`	
`4 - 5`	`-1`	
`2 + 7.3`	`9.3`	Il risultato è un numero reale perchè uno dei due operandi è un numero reale, da notare l'utilizzo del punto anzichè della virgola
`4.5 * 3.2`	`14.4`	Il simbolo della moltiplicazione è $*$ anzichè $\times$
`12 / 4`	`3`	
`13 / 4`	`3`	La divisione tra due numeri interi da come risultato un numero intero, quindi è come se venisse eliminata la parte dopo la virgola
`8 / 9`	`0`	Per quanto già detto nell'esempio precedente il risultato di questa operazione è 0
`13 / 4.0`	`3.25`	Uno dei due operandi è un numero reale, la divisione avrà anche la parte decimale
`1 + 2 * 3`	`7`	La moltiplicazione ha precedenza maggiore della somma
`(1 + 2) * 3`	`9`	Le parentesi cambiano le precedenze
`((1 + 2)*(8 - 3) + 7)*3`	`66`	Le parentesi possono essere annidate
`14 % 5`	`4`	Resto della divisione intera, ha senso solo se entrambi gli operandi sono interi
`16 % 4 == 0`	`0`	Viene spesso usato per verificare se un numero è multiplo di un altro, controllando se il resto risulta 0. In questo esempio la risposta sarebbe positiva.

A.2. Operatori relazionali

Gli operatori *relazionali*, a volte chiamati *di confronto*, sono quelli che vengono utilizzati quando bisogna confrontare tra loro due valori tra i quali esiste una relazione d'ordine, quindi ad esempio due numeri, e che servono a rispondere a domande del tipo "A è minore di B?". Gli operatori relazionali sono quelli noti, riassunti nella Tabella A.3.

Da notare che l'operatore di uguaglianza nei linguaggi che verranno proposti nel seguito viene indicato con i due simboli == anzichè con il più familiare =, che, come già visto, è invece riservato all'assegnamento. In tutti gli operatori che sono composti da due simboli, l'ordine è importante: scrivere <= è diverso da scrivere =>, il primo è corretto, il secondo no.

Operazione	Simbolo
Maggiore di	`>`
Minore di	`<`
Maggiore o uguale a	`>=`
Minore o uguale a	`<=`
Uguale a	`==`
Diverso da	`!=`

Tabella A.3.: Operatori matematici

Le espressioni che contengono questi operatori vengono solitamente usate come condizioni nelle selezioni o nelle iterazioni, in quanto possono assumere il valore **vero** (`true`) o **falso** (`false`) e quindi determinare quale azione intraprendere nelle successive parti del codice.

Tabella A.4.: Esempio di utilizzo degli operatori relazionali

Operazione	Risultato	Note
`2 < 7`	`true`	
`2 > 7`	`false`	
`13.4 <= 24`	`true`	Si possono confrontare tra loro anche numeri di tipi diversi, il risultato è quello atteso
`25 >= 25`	`true`	
`42 == 42`	`true`	
`42 == 43`	`false`	
`42 != 43`	`true`	

A.3. Operatori logici

Gli operatori *logici* sono quelli dell'algebra di Boole, nella quale esistono solo due simboli, il *vero* e il *falso* e si combinano tra loro attraverso gli operatori fondamentali[2] *OR*, *AND* e *NOT*.

2: Ne esistono anche altri, ma questi tre sono sufficienti per rappresentare qualsiasi tipo di espressione logica

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

Tabella A.5.: Operatori logici

Scopo di questi operatori è mettere in relazione due o più condizioni e verificare la verità o falsità dell'espressione complessiva. Generalmente in italiano l'operatore AND assume il significato della *e* congiunzione, l'operatore OR viene rappresentato dalla *o* come viene usata nell'espressione "o l'una o l'altra", mentre il NOT corrisponde al *non* delle negazioni. Se ad esempio si avessero come condizioni "**NON** avere mai preso un voto inferiore a 25"

e “Avere una media dei voti degli esami di almeno 27", si potrebbero connettere utilizzando un AND, ottenendo la frase “**NON** avere mai preso un voto inferiore a 25 **E** avere una media dei voti degli esami di almeno 27, permette di continuare a frequentare la Scuola Galileiana", che quindi indica la possibilità di frequentare solo se **entrambe** le condizioni sono vere. Viceversa se si volesse utilizzare l’operatore OR allora la frase diventerebbe “**NON** avere mai preso un voto inferiore a 25 **O** avere una media dei voti degli esami di almeno 27, permette di continuare a frequentare la Scuola Galileiana", indicherebbe invece che si può frequentare a patto che **almeno** una delle due condizioni sia vera.

Nella Tabella A.6 si possono trovare riassunte tutte le possibili combinazioni che si possono ottenere con questi operatori logici.

Tabella A.6.: Tabelle di verità

A	B	A OR B	A AND B	NOT A
`false`	`false`	`false`	`false`	`true`
`false`	`true`	`true`	`false`	`true`
`true`	`false`	`true`	`false`	`false`
`true`	`true`	`true`	`true`	`false`

Questi operatori possono essere utilizzati ad esempio per definire l’appartenza o meno di una variabile a un certo intervallo, piuttosto che verificare che alcune variabili soddisfino *contemporaneamente* certe condizioni, come si può vedere negli esempi di Tabella A.7.

Tabella A.7.: Esempio di utilizzo degli operatori logici

Operazione	Risultato
`a > 5 && a < 17`	`true` se `a` è compresa tra 5 e 17, estremi esclusi
`a < 5 || a > 17`	`true` se `a` si trova all’esterno dell’intervallo (5, 17)
`i < 10 && j < 15`	`true` se `i` è minore di 10 e contemporaneamente `j` è minore di 15
`n % 3 == 0 && !(n % 6 == 0)`	`true` se `n` è divisibile per 3 ma non è divisibile per 6

A.4. Funzioni C++ di uso frequente

Le funzioni servono a svolgere dei compiti più o meno elementari che però non sono implementati direttamente nell’hardware del computer. Ogni linguaggio ha un insieme di funzioni di utilizzo comune che solitamente è composto da centinaia di elementi: in questo paragrafo verranno mostrate solo alcune delle funzioni del C++ di utilizzo più frequente, rimandando alla ricerca su Internet per eventuali altre.

Ognuna di questo funzioni si trova all’interno di un *header* di libreria, che quindi va incluso prima del suo utilizzo. Non è

Tabella A.8.: Funzioni di utilizzo comune

Nome	Esempio	Note
`sqrt`	`sqrt(17)`	Estrae la radice quadrata del numero o della variabile contenuta nelle parentesi tonde, in questo esempio produrrà 4.12310
`abs`	`abs(-3)`	Produce il valore assoluto del numero o della variabile contenuta nelle parentesi tonde, in questo esempio produrrà 3
`round`	`round(3.78)`	Produce il numero intero più vicino a quello passato come parametro, in questo esempio produrrà 3. Si ricorda invece che il semplice assegnamento di un numero con virgola a un intero effettua il troncamento della parte decimale.
`rand`	`rand()`	Produce un numero intero casuale compreso tra 0 e un numero che dipende dal compilatore utilizzato, nel caso più comune vale 32768 (2^{15})
`rand`	`rand() % n`	Produce un numero intero casuale compreso tra 0 e n - 1: questa scrittura viene utilizzata per limitare i valori prodotti all'interno dell'insieme [0, n) (0 incluso, n escluso)
`srand`	`srand(time(NULL))`	Inizializza il generatore di numero casuali in modo che la funzione precedente produca numeri diversi ad ogni esecuzione di un programma. Va chiamata **una volta sola** all'inizio del programma.

possibile stabilire in maniera certa quale sia l'header relativo ad ogni funzione, poiché dipendente dall'implementazione del compilatore: per queste funzioni, comunque, solitamente l'header da includere è `cstdlib`.