Appunti di
INTRODUZIONE ALLA PROGRAMMAZIONE[1]

Francesca Odone

Anno Accademico 2021-2022

[1]Materiale basato su note dei prof G. Chiola, G. Costa, E. Puppo - 2007-2013

Obiettivi di questo documento

Questi appunti completano le slide presentate in classe e forniscono un ulteriore supporto, aggiuntivo rispetto ai testi consigliati, andando a colmare alcune lacune presenti nei testi classici di introduzione alla programmazione. Aiuteranno lo studente a mettere in relazione parti diverse del corso e costruire legami con altri corsi. Gli appunti contengono informazioni aggiuntive e approfondimenti che non sono coperti dalle lezioni, ma che hanno lo scopo di completare il quadro d'insieme che il corso mira a fornire. Infine questi appunti non seguono necessariamente l'ordine cronologico delle lezioni, per avere una linea guida in questo senso si consiglia di consultare le slide.

Parte 1

Introduzione alla Programmazione

Il termine “programmazione” fa riferimento all’insieme delle attività che portano allo sviluppo di applicazioni o programmi (“software” in inglese) eseguibili su un sistema di calcolo per risolvere un problema predeterminato. Le attività della programmazione si dividono in due parti principali, connesse tra loro (Figura 1.1): la fase di *problem solving*, durante la quale traduciamo le specifiche del problema dato in una sequenza di passi fondamentali atti a risolverlo (algoritmo); la fase di *implementazione*, durante la quale traduciamo l’algoritmo in un programma scritto in uno specifico linguaggio di programmazione. Le due fasi sono egualmente importanti, ma spesso l’attività di problem solving viene sottovalutata. Negli ultimi anni questa fase ha acquisito una nuova importanza anche agli occhi dei non esperti, quando si è iniziato a parlare di *pensiero computazionale* e delle sue implicazioni sull’insegnamento della programmazione.

Il pensiero computazionale coinvolge una serie di abilità di problem-solving che sono tipiche dell’attività di programmazione, ma che possono essere applicatiead altre discipline. Il pensiero computazionale include specifiche tecniche:

- *Decomposizione*: l’abilità di spezzare un problema in “passi atomici”, così da poterlo spiegare a un altro agente (persona o computer).
- *Riconoscimento di pattern:* l’abilità di notare similitudini o differenze comuni che ci permettono di fare previsioni o ci portano a scorciatoie.
- *Generalizzazione e astrazione*: l’abilità di filtrare informazioni non necessarie e generalizzare quelle che invece sono utili. Ci permettono di rappresentare un’idea o un processo in termini generali, così da poterlo utilizzare per risolvere problemi simili.
- *Progettazione di algoritmi* : l’abilità di sviluppare una strategia passo-passo per risolvere un problema.

Una volta ricevute le specifiche di un problema, messe a punto le tecniche del pensiero computazionale che ci portano ad analizzare e comprendere il problema, dividerlo in componenti più semplici ognuna delle quali possa essere risolta con una sequenza di passi, inizieremo a pensare ad una traduzione in un linguaggio di programmazione (e quindi otterremo un programma) .

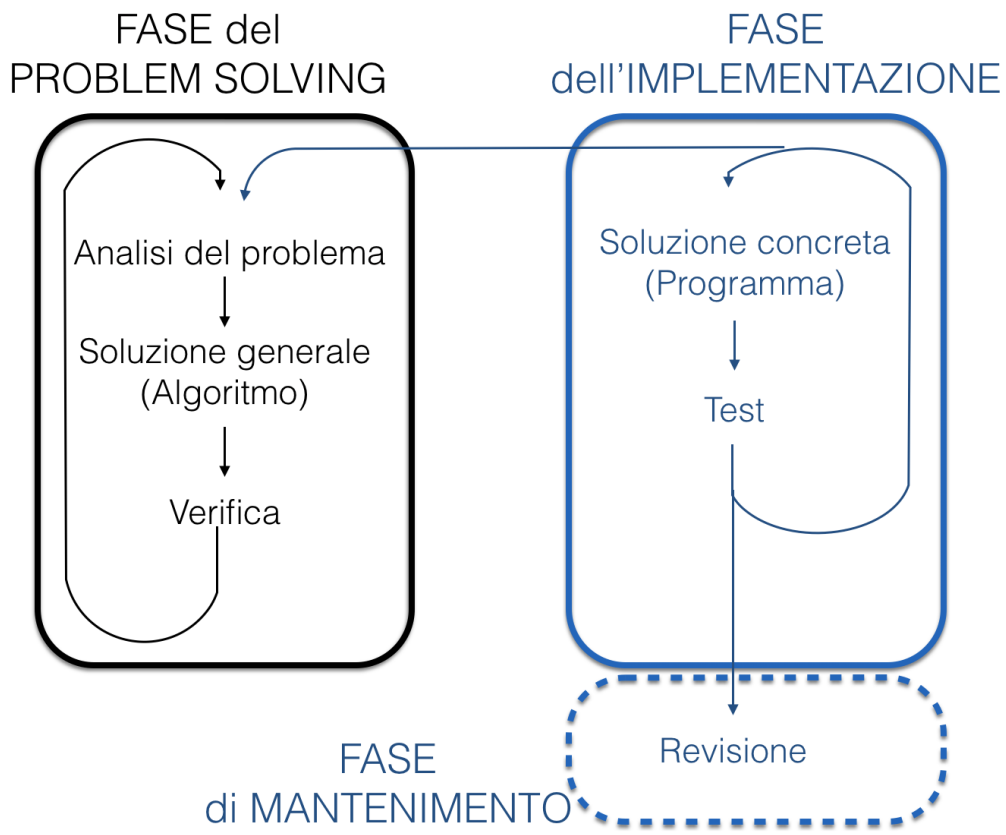


Figura 1.1: Il ciclo di vita del software

Le principali caratteristiche di un "buon programma" possono essere riassunte in:

- rispondenza alle specifiche (l'applicazione deve svolgere esattamente il compito che il committente del programma ha stabilito, nulla di meno, nulla di diverso, e possibilmente anche nulla di più);
- correttezza e completezza (l'applicazione non deve mai dare risultati sbagliati e deve essere sempre in grado di portare a termine il calcolo richiesto, a meno di guasti del sistema di calcolo su cui "gira");
- efficienza ed economia nell'uso delle risorse (l'applicazione non deve occupare quantità eccessive di memoria o richiedere tempi di esecuzione troppo lunghi);
- scalabilità (l'applicazione deve poter operare su problemi di dimensione diversa, sia più piccoli che più grandi del "caso normale" preso a riferimento nella fase di sviluppo del programma);
- robustezza, affidabilità, tolleranza ai guasti (può sempre succedere che una o più componenti di un sistema di calcolo, hardware o software, si guasti o sia soggetta a malfunzionamenti temporanei; anche in questi casi è auspicabile che la nostra applicazione non fallisca del tutto, ma sia comunque in grado di funzionare in modo predicibile, magari portando a calcolare risultati parziali ma non errati, eventualmente a scapito della sola efficienza nell'uso delle risorse).

Lo sviluppo di un programma che abbia tutte (o buone parte) delle caratteristiche sopra elencate è molto difficile, e richiede un grande dispendio di tempo ed energie da parte di "programmatori" molto preparati. Tale difficoltà cresce in modo non lineare con la "lunghezza" del programma, per cui è prassi aspettarsi che un programma lungo e complesso sia inevitabilmente molto problematico da sviluppare, anche da parte di programmatori molto esperti e qualificati. Nel caso di programmi sufficientemente "piccoli", invece, ci si aspetta che un "buon programmatore" sia in grado di raggiungere agevolmente l'obiettivo, mentre un "programmatore mediocre" potrebbe anche in questo caso trovare delle serie difficoltà nel portare a termine il compito a lui assegnato.

Cerchiamo allora di capire cos'è che può fare la differenza tra un "buon" programmatore ed uno "mediocre". Il libro "storico" (una vera e propria opera enciclopedica in vari volumi) sull'argomento è il testo del professor Donald Knuth che si intitola: "The art of computer programming". Questo titolo evidenzia molto bene l'aspetto "creativo" del programmatore che si contrappone all'idea dominante di una disciplina fatta di formalizzazioni matematiche e aspetti nozionistici. Oltre alle capacità di ragionamento logico/deduttivo, quali sono le doti naturali del "buon programmatore"? Un po' scherzosamente (ma neanche tanto) possono essere riassunte con:

- astuzia, pigrizia e diligenza (tener conto di tutti i dettagli ma cercando sempre il modo più astuto per minimizzare lo sforzo; fare tutto e solo quello che viene commissionato e mai nulla di più);
- impazienza e tenacia (cercare di arrivare il prima possibile ad una soluzione funzionante, magari in un caso semplificato, ma senza desistere se le cose vanno per le lunghe, ed avendo ben chiaro fin dall'inizio come estendere il programma in modo che possa rispondere a tutte le specifiche del committente);
- arroganza e umiltà (conoscere i propri limiti ma scommettere sulla propria capacità di superarli, anche se l'obiettivo è molto ambizioso; l'arroganza porta spesso a commettere errori, ma l'umiltà permette di riconoscerli e imparare a non ripeterli).

Come si vede ogni qualità richiesta va opportunamente bilanciata da quella apparentemente opposta, ed il giusto equilibrio si trova di solito solo dopo anni di esperienza sul campo.

Consci di questo, possiamo quindi affermare che scopo del corso di Introduzione alla Programmazione non è tanto quello di "insegnare a programmare" (sarebbe un obiettivo non realistico per un corso di questa durata) quanto quello di fornire quelle informazioni e metodologie di base che possono servire poi, col tempo, a sviluppare la capacità di programmare con la propria esperienza e sfruttando eventualmente il proprio talento naturale (se c'è).

Cominceremo quindi introducendo concetti semplici ma fondamentali quali il ciclo di vita del software e ci soffermeremo in particolare sugli elementi di analisi del problema e di ragionamento computazionale. Subito dopo esploreremo gli elementi principali di un sistema di calcolo, senza dimenticare che il corso di Architetture, che si svolge parallelamente ad Introduzione alla Programmazione, fornirà approfondimenti in questo senso. Successivamente, attraverso l'uso di un linguaggio di programmazione molto diffuso (il C++, ovvero un suo sottoinsieme), le principali caratteristiche della *programmazione imperativa,* che rappresenta lo stile di programmazione maggiormente utilizzato e più semplice da apprendere. Infine passeremo allo studio degli algoritmi più semplici e comuni per la soluzione di alcuni problemi di programmazione classici.

E' importante iniziare presto un percorso di approfondimento delle tematiche che possa poi portare, col tempo, a sviluppare la capacità di analizzare un problema, ricavarne le specifiche e programmarne la soluzione in modo corretto, rapido ed efficiente. Il corso di Algoritmi e Strutture dati sarà la naturale prosecuzione di questo percorso, presentando problemi più complessi e soluzioni più sofisticate. Il corso di Architetture permetterà di approfondire la parte iniziale sulla struttura dei sistemi di calcolo, sia a livello hardware che a livello di sistema operativo (propedeutica tra l'altro ad una comprensione della efficienza e complessità dei programmi). Il corso di Linguaggi di Programmazione Orientata agli Oggetti e poi altri corsi del terzo anno o - per chi proseguirà - della Laurea Magistrale, presenteranno altri paradigmi e tecniche di programmazione di più alto livello, che permettono di dominare più facilmente problemi ed applicazioni di complessità crescente.

Nel proseguire il percorso di studi scoprirete che le vostre attività di programmatori non rappresenteranno più solo un obiettivo a sé (imparare un nuovo linguaggio, impadronirsi di una tecnica di programmazione più raffinata, ...), ma diventeranno un bagaglio di conoscenze di base che *vi aprirà la porta a nuove discipline*, per le quali il saper programmare è requisito irrinunciabile. Ne citiamo alcune, senza l'ambizione di essere esaustivi: l'intelligenza artificiale, l'analisi dei segnali e delle immagini, la cyber-security, la computer graphics, ...

Parte 2

Sistemi di calcolo

Una delle caratteristiche peculiari dei cosiddetti "sistemi di calcolo", che ne ha accompagnato l'evoluzione sin dalle origini negli anni della seconda guerra mondiale, è la loro grande *complessità*. Tale caratteristica si riperquote in diverse attività di un "informatico" quali la progettazione, la realizzazione, l'uso, la comprensione e l'insegnamento dell'uso di un sistema di calcolo.

Al fine di poter trattare sistemi molto complessi, gli informatici hanno sviluppato una serie di "contromisure" (che vedremo man mano andando avanti nel corso), tra le quali la più efficace è senza dubbio la *scomposizione (o strutturazione) in diversi livelli di astrazione*. Tale scomposizione viene ottenuta normalmente attraverso l'introduzione dei concetti di *"macchina virtuale", "linguaggio (formale)"* e di *"codifica delle informazioni"*.

Si possono dare definizioni matematiche rigorose di tali concetti. Definiamo l'*alfabeto* di una macchina virtuale come un insieme finito di simboli diversi tra loro riconoscibili ed utilizzabili dalla macchina virtuale. Il *linguaggio* della macchina virtuale è definito come l'insieme di tutte le sequenze di simboli dell'alfabeto che identificano comandi eseguibili dalla macchina virtuale oppure dati usati o prodotti dalla macchina virtuale durante l'esecuzione dei comandi. Il linguaggio rappresenta quindi lo strumento formale con il quale vengono codificate le informazioni: sia i *dati,* ossia la "materia" che verrà manipolata, che i *comandi,* ossia l'insieme delle possibili azioni che possono essere compiute per manipolare i dati. La *macchina virtuale* è quindi un dispositivo in grado di *interpretare* il linguaggio, ossia di manipolare i dati eseguendo i comandi del linguaggio stesso. Una macchina virtuale è tale rispetto ad un determinato linguaggio ed il fatto che sia in grado di interpretare quel linguaggio la rende tale indipendentemente da come tale macchina viene realizzata fisicamente. Viene quindi detta *virtuale* perché non è necessariamente associata ad una determinata "macchina" reale: la stessa macchina virtuale potrebbe essere realizzata mediante diversi dispositivi, oppure essere una persona, o un insieme di persone, o un insieme anche complesso di macchine e persone.

2.1 Macchine reali e virtuali

Prima di addentrarci nello studio dei sistemi di calcolo con l'ausilio del concetto di macchina virtuale è utile fare qualche considerazione "di buon senso" basata sulla esperienza quotidiana di ciascuno di noi. È infatti importante comprendere che l'informatica (che è la scienza che studia i sistemi di calcolo, tant'è che in inglese venne coniato il termine di "computer science" per definirla) non è estranea alla vita quotidiana non tanto e non solo perchè una quota sempre maggiore di elettrodomestici e strumenti di uso comune fanno uso di tecnologie informatiche, ma anche e sopratutto perchè molto spesso noi stessi ragioniamo in termini informatici (spesso senza neanche rendercene conto). Una presa di coscienza esplicita della logica del nostro comportamento razionale ed una sua formalizzazione in linguaggio informatico costituisce quindi il punto di partenza per una comprensione più profonda e per una maggior consapevolezza del mondo in cui viviamo.

2.1.1 La lavabiancheria

Cominciamo quindi a considerare un esempio dalla vita domestica: l'uso di una lavabiancheria per il bucato. Indipendentemente dalla presenza di un "microprocessore" all'interno della macchina, possiamo subito individuare una peculiarità di questo elettrodomestico: la sua capacità di potersi adattare a diversi tipi di tessuti da lavare ed a diversi livelli di risultati in termini di bucato prodotto e di esigenze di tempo di esecuzione e/o di consumo di risorse.

Tipicamente, dopo aver inserito il bucato ed il detersivo, dobbiamo girare una o più manopole (oppure premere pulsanti) per la scelta del tipo (*programma*) di lavaggio desiderato. Alternative tipiche possone essere: "Cotone", "Colorati", "Sintetici", "Lana", ecc. Tali nomi formano ciò che in termini informatici viene detto un "linguaggio" (nel caso specifico si tratta di un linguaggio degenere che si riduce ad un semplice vocabolario di termini, ossia un "lessico") mediante il quale noi possiamo *impartire ordini* che verranno eseguiti alla lettera dalla macchina, senza alcun controllo "intelligente". Per esempio, se inseriamo nella macchina una maglietta di lana con colori delicati, e programmiamo l'attivazione di un ciclo di lavaggio adatto per lenzuola bianche, la lavabiancheria eseguirà l'ordine ricevuto riscaldando l'acqua alla temperatura di 60 gradi, con risultati disastrosi. È quindi onere/responsabilità dell'utilizzatore impartire alla macchina ordini coerenti utilizzando il linguaggio di programmazione descritto sul manuale d'uso.

La comprensione del manuale d'uso e l'uso della lavabiancheria prescindono da come la macchina realizza effettivamente il lavaggio. Quando la nostra macchina si guasta, il tecnico della manutenzione che viene ad effettuare la riparazione userà delle descrizioni dello stesso elettrodomestico diverse dal manuale d'uso (schemi elettrici, catalogo delle parti di ricambio, ecc.) per portare a termine la riparazione, evidenziando quindi un modo totalmente diverso di osservare e manipolare la stessa macchina. Il tecnico che effettua la manutenzione vede quindi la nostra lavatrice ad un livello di astrazione diverso dal nostro, e questo a volte complica la comunicazione tra il tecnico ed il cliente, ma certamente semplifica la vita ad entrambi nella loro interazione con la macchina.

2.1.2 La produzione di pizze

Le metodologie informatiche si applicano quotidianamente non solo all'interazione con gli elettrodomestici ma anche in tante circostanze dove non si usano vere e proprie macchine fisiche programmabili. Anche in questi casi, però, possiamo ricorrere al concetto di "macchina virtuale" per studiare la situazione.

Passiamo quindi ad un esempio forse più complesso e certamente meno associabile all'uso di computer rispetto a quello della lavatrice: il problema di procurarsi una pizza per mangiare. Possiamo anzitutto scegliere tra due alternative: cucinare la pizza in casa, oppure recarci in una pizzeria.

Cominciamo ad esaminare la prima alternativa. Per cucinare la pizza in casa occorre procurarsi gli ingredienti (acqua, farina, lievito, ecc.), la ricetta (per esempio da un libro di cucina), ed una serie di attrezzi per la realizzazione della ricetta (mattarello, tavolo, forno, ecc.). Le attività necessarie per procurarsi ingredienti ed attrezzi possono a loro volta essere descritti in termini di macchine virtuali che risolvono altri problemi (procurarsi la farina, ecc.) che supponiamo per il momento di poter trascurare (per esempio chiedendo a qualcun altro di risolverli al nostro posto). Tale procedimento di scomposizione del problema originario in fasi diverse, delega di alcune fasi "ad altri" e focalizzazione della nostra attenzione su una fase particolare, è un altro tipico esempio di metodo applicato dagli informatici per affrontare problemi complessi, chiamato *"scomposizione funzionale"*.

Una volta procurati tutti gli ingredienti e gli attrezzi (da parte nostra o di qualche nostro collaboratore) possiamo passare alla fase di preparazione vera e propria della pizza, seguendo i passi descritti nella ricetta. La ricetta farà uso di un linguaggio che noi dobbiamo essere in grado di comprendere correttamente e dovrà descrivere il procedimento di preparazione in termini di passi elementari che noi dobbiamo essere in grado di realizzare. Se, per esempio, la ricetta fosse scritta in inglese noi dovremo sapere che il termine "tomato sauce" indica la passata di pomodoro. D'altra parte, se leggendo la ricetta incontriamo l'istruzione "scaldare il forno a 180 gradi", dobbiamo essere in grado di accendere il forno e portarlo alla temperatura corretta; le procedure possono essere diverse per un forno a legna, a gas o elettrico, e non è affatto detto che una persona in grado di scaldare un forno elettrico sia capace di scaldare correttamente anche un forno a legna, nel qual caso avrebbe bisogno di istruzioni più dettagliate per poter portare a termine correttamente l'operazione.

In termini informatici, la realizzazione di una pizza seguendo passo passo le indicazioni di una ricetta costituisce una attività detta *interpretazione* da parte di una macchina virtuale (noi stessi nella fattispecie). L'eventuale interruzione della interpretazione della ricetta al momento del riscaldamento del forno per andare a leggere su un altro manuale le istruzioni dettagliate di accensione e regolazione della temperatura, in termini informatici costituisce una *estensione procedurale* per sopperire ad una mancanza di corrispondenza precisa tra le istruzioni contenute nella ricetta (*programma* da eseguire) e l'insieme delle istruzioni che la macchina virtuale (noi stessi) conosce ed è in grado di eseguire direttamente come azioni elementari. Al termine dell'interpretazione della procedura "scaldare il forno" la macchina virtuale può tornare all'interpretazione del programma principale, dal punto in cui l'aveva interrotto.

La seconda alternativa (andare in pizzeria) può risultare economicamente più dispendiosa ma anche più comoda, in quanto qualcun altro svolgerà il ruolo di macchina virtuale al posto nostro (a fronte di un compenso in denaro). Normalmente, in pizzeria l'interazione avviene con un cameriere, il quale assume agli occhi del

cliente il ruolo di macchina virtuale.

Inizialmente, il linguaggio usato per la comunicazione tra cliente e cameriere sarà quello dei nomi delle pizze scritte sul menù ("margherita", "quattro stagioni", "napoletana", ecc.). Tale linguaggio può essere considerato ad *alto livello* in quanto ogni parola corrisponde direttamente con una particolare esigenza dell'utilizzatore della macchina, piuttosto che ad una esigenza di semplicità o di efficienza da parte della macchina stessa. Ordinando al cameriere "una quattro stagioni" noi comandiamo alla macchina virtuale pizzeria l'esecuzione di una attività complessa, il cui risultato è quello di far arrivare al nostro tavolo la pizza desiderata.

L'utente non è normalmente interessato ai dettagli di realizzazione della pizza, per cui si pone ad un livello di astrazione superiore e percepisce l'attività della macchina virtuale "pizzeria" come un tutt'uno che termina con il recapito della pizza desiderata al proprio tavolo dopo una certa quantità (possibilmente breve) di tempo. Il cameriere si pone invece ad un livello di astrazione inferiore per poter *realizzare (implementare)* la macchina virtuale con le funzionalità richieste dal cliente sulla base delle macchine virtuali di cui lui stesso può disporre.

Tipicamente il cameriere scriverà l'ordine su un pezzo di carta col duplice scopo di passare l'ordine al pizzaiolo e di tener traccia della consumazione per poter poi presentare il conto. Pizzerie più tecnologicamente evolute potranno far uso di palmari in dotazione ai camerieri con collegamenti wireless con la cucina e con la cassa, ma da un punto di vista concettuale l'approccio non cambia rispetto alla versione cartacea/tradizionale. Normalmente, per risparmiare tempo, il cameriere userà delle convenzioni per *codificare* in forma abbreviata l'ordine ricevuto dal cliente; per esempio qualcuno potrebbe scrivere "1x4S" (al posto di "una quattro stagioni"), qualcun altro potrebbe scrivere "Q.S.", qualcun altro ancora potrebbe scrivere "2" (se il nome "quattro stagioni" compare al secondo posto nella lista dei nomi del menù, come nel nostro esempio, dove 1 significherebbe "margherita", e 3 corrisponderebbe a "napoletana").

Una volta trascritto l'ordine, il cameriere lo passa al pizzaiolo e può dedicarsi al servizio di un altro tavolo, demandando quindi il completamento dell'esecuzione dell'ordine ad un'altra macchina virtuale (un produttore di automobili chiamerebbe questa tecnica "catena di montaggio", mentre noi informatici preferiamo usare il termine inglese *pipeline*). La macchina virtuale pizzaiolo interpreta gli ordini ricevuti dal cameriere mediante applicazione di una serie di passi elementari che ha imparato ad eseguire una volta per tutte e che ricorda in permanenza senza bisogno di far ricorso ogni volta alla lettura di ricette di cucina (stendere la pasta, aggiungere la passata di pomodoro, aggiungere la mozzarella, ecc.). In termini informatici, il pizzaiolo esegue direttamente una istruzione della sua macchina virtuale traducendola in una sequenza predefinita di azioni elementari memorizzate in una sua memoria permanente (*firmware*).

2.2 Struttura dei sistemi di calcolo

Storicamente si è venuta consolidando una stratificazione dei livelli di astrazione utilizzati per progettare e per descrivere le funzionalità di un sistema di calcolo secondo il seguente schema:

L5: Linguaggi di alto livello (Pascal, Java, C, C++, ecc.).
È il livello di macchina virtuale normalmente usato dai programmatori di applicazioni informatiche,

oggetto principale di studio nel corso di Programmazione e di molti altri corsi successivi di informatica.

L4: Assembler e librerie (Assembler, binding, threads, run-time, ecc.).
È il livello di macchina virtuale più basso utilizzabile dal programmatore di applicazioni; in pratica, si usa solo in situazioni particolari, di solito legate alla necessità di ottenere prestazioni in tempo reale. Si vedranno esempi in vari corsi, tra cui Sistemi di elaborazione dell'informazione.

L3: Nucleo del sistema operativo (Linux, processi, driver, ecc.).
È il livello di macchina virtuale che permette l'attivazione di programmi indipendenti (processi) e l'uso delle risorse fisiche del sistema da parte di questi programmi, senza interazioni logiche tra loro; verrà studiato prevalentemente nei corsi di Sistemi di elaborazione dell'informazione.

L2: Macchina convenzionale (microprocessori).
È il livello di definizione delle istruzioni base del computer e degli altri dispositivi fisici che compongono il sistema. Verrà studiato prevalentemente nel corso di Sistemi di elaborazione dell'informazione, con qualche anticipazione in questo corso.

L1: Microarchitettura (trasferimento tra registri, memorie, bus, ecc.).
È il livello di definizione del funzionamento dei singoli componenti fisici del sistema in termini di interconnessione e spostamento di informazioni tra circuiti logici elementari; determina le caratteristiche di velocità, capacità di memorizzazione e di comunicazione dei dispositivi, e sarà oggetto principale di studio nel corso di Sistemi di elaborazione dell'informazione.

L0: Logica circuitale (circuiti combinatori, sequenziali, sincroni, ecc.).
È il livello di realizzazione dei circuiti logici elementari basato sulla logica Booleana. Verrà trattato nel corso di Sistemi di elaborazione dell'informazione.

L-1: Elettronica/fotonica (transistors, led, laser, ecc.).
È il livello di progettazione dei dispositivi fisici, e non sarà affrontato nel corso di Laurea in Informatica. Questi aspetti sono trattati nell'ambito dell'Ingegneria Elettronica.

L-2: Fisica dello stato solido (semiconduttori, quantistica, ecc.).
È il livello usato per progettare e realizzare i circuiti integrati integrati da parte dei produttori e non sarà studiato nel corso di Laurea in Informatica. Questi aspetti dovrebbe sono trattati nell'ambito della Fisica.

Ciascun livello elencato nella tabella può essere formalizzato ed analizzato come una macchina virtuale dotata di un proprio linguaggio a cui corrisponde un insieme di istruzioni eseguibili. Normalmente le istruzioni realizzate da una macchina virtuale di livello Li (dove i rappresenta un numero intero tra 0 e 5) comportano una realizzazione diretta mediante istruzioni di una macchina virtuale di livello Lj (dove j è strettamente minore di i, ossia in termini matematici $j < i$), oppure una interpretazione in termini di un insieme di istruzioni di una macchina virtuale di livello inferiore Lj, oppure una estensione procedurale di un insieme di istruzioni allo stesso livello Li.

Ragionando in termini puramente astratti possiamo quindi concludere che ogni istruzione del livello L5 (per esempio ogni istruzione del linguaggio C++ che vedremo durante il corso) può essere descritta in termini

di sequenze di istruzioni del livello L1. Questo è vero anche in termini pratici: potremmo pensare di monitorare e registrare il comportamento dei dispositivi fisici che compongono il sistema di calcolo al livello L1, ed associare una serie complessa di attività all'esecuzione di una singola istruzione di livello L5. Concettualmente potremmo quindi arrivare a concludere che la stratificazione proposta nella tabella non ha motivazioni teoriche per esistere, in quanto i programmi potrebbero essere scritti dal programmatore direttamente usando il linguaggio della macchina virtuale L1, invece del C++. Questo tentativo di "semplificazione" del problema sarebbe però destinato a fallire miseramente in quanto non tiene conto di un vincolo fondamentale: il limite intrinseco alle capacità cognitive del programmatore destinato ad usare la macchina virtuale "sistema di calcolo". Nessuno di noi riuscirebbe a raccapezzarsi nella descrizione di alcune decine o centinaia di migliaia di istruzioni elementari del livello di astrazione L1 corrispondenti a poche istruzioni di un programma C++ per la somma di due numeri interi e la stampa del risultato.

Scopo della stratificazione in livelli di astrazione è dunque quello di colmare il divario (*semantic gap*) tra il modo di pensare alla soluzione dei problemi da parte del programmatore ed il modo di realizzarla delle macchine che, sfruttando fenomeni fisici conosciuti, siano in grado di eseguire il programma che porta alla soluzione del problema. Tale tecnica viene applicata più volte nel caso di soluzione di problemi complessi: anche lavorando con una macchina virtuale di livello L5 il programmatore può sentire la necessità (o sfruttare l'opportunità) di definire un'ulteriore stratificazione in livelli di astrazione diversi, a cui corrisponde una *pila* di macchine virtuali destinate a colmare il divario tra le istruzioni fornite dal linguaggio di programmazione e il suo modo di risolvere concettualmente un problema complesso.

2.3 Macchine convenzionali

In pratica, la programmazione coinvolge prevalente una macchina virtuale di livello 5, e quindi questo sarà il livello di astrazione sul quale ci soffermeremo maggiormente in questo corso introduttivo alla programmazione, in particolare facendo riferimento alla macchina virtuale definita dal linguaggio di programmazione "C++". Tuttavia, per capire alcune peculiarità di questa macchina virtuale è utile conoscere qualcosa anche dei livelli L4 ed L2 sulle quali una realizzazione efficiente della macchina "C++" è solitamente basata. Nel suggerirvi di fare riferimento all'insegnamento di Architetture dei Calcolatori, per trovare una trattazione dettagliata e rigorosa di questi livelli di astrazione, ci soffermiamo qui su alcuni concetti elementari e su alcuni esempi. Nel fare ciò prenderemo spunto dalla effettiva evoluzione storica dei primi sistemi di calcolo tra il 1940 e il 1960.

2.3.1 Macchina di Von Neumann

Facciamo qui riferimento ad una organizzazione di un sistema di calcolo, la Macchina di Von Neumann, nel seguito indicato con MVN, effettivamente realizzato e reso operativo nell'ambito di un progetto dell'Università di Princeton sotto la guida del matematico John Von Neumann nel periodo della seconda guerra mondiale. Prescinderemo ovviamente da alcuni dettagli operativi derivanti dai limiti della tecnologia elettronica degli anni '40, oggi ampiamente superati, mentre ci concentreremo sulle intuizioni di base, che ancor oggi stanno dietro la progettazione e la realizzazione dei sistemi di calcolo.

La macchina venne realizzata mediante l'interconnessione di quattro dispositivi complessi: memoria RAM, unità operativa, ingresso e uscita, come illustrato in Figura 2.1.

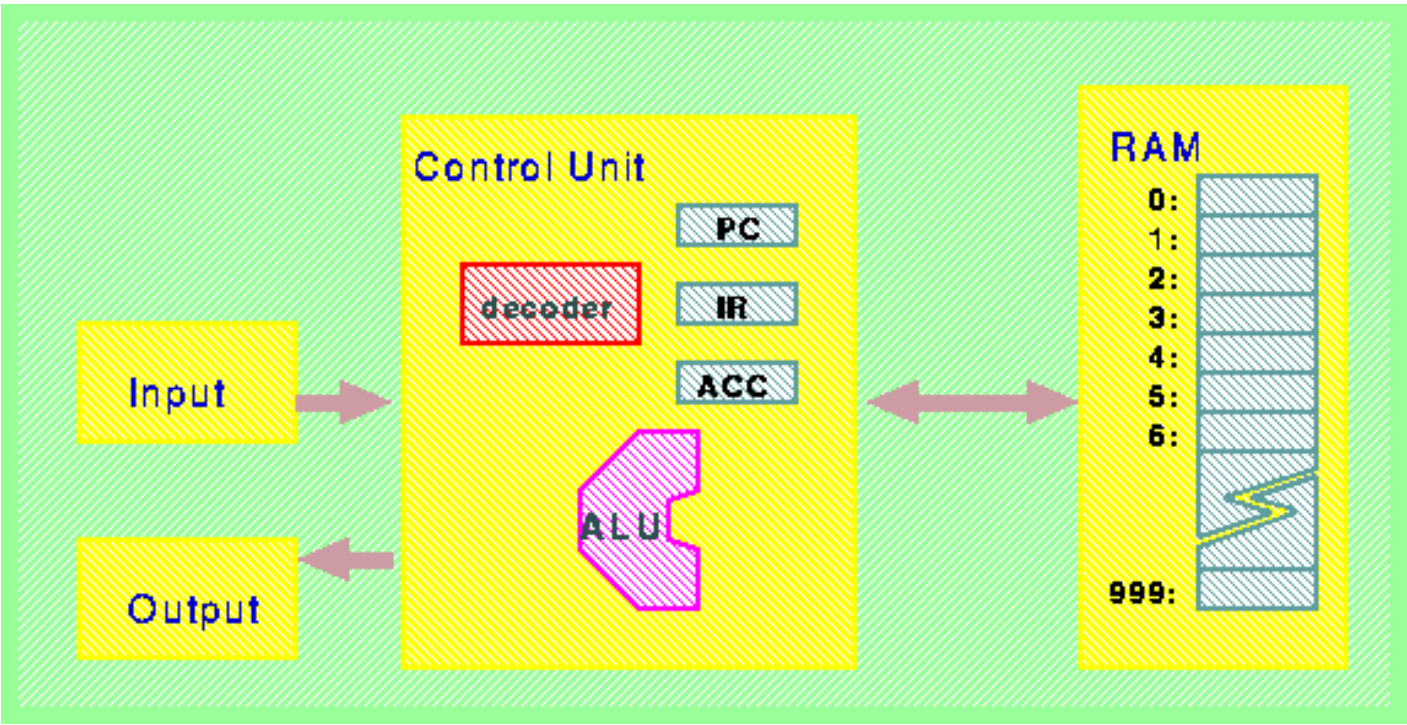


Figura 2.1: Architettura Von Neumann

I quattro componenti in una riproduzione moderna della MVN oggi potrebbero essere definiti come segue:

Input (dispositivo di ingresso)
Permette all'utente di interagire con la macchina (per esempio attraverso l'uso di una tastiera numerica tipo quella di un telefono) per l'introduzione di numeri interi, uno alla volta, una cifra decimale dopo l'altra fino a quando non si preme il tasto "invio". Eventualmente la presenza di un tasto di "cancellazione" potrebbe permettere di correggere errori di battitura prima dell'invio del numero.

Output (dispositivo di uscita)
Permette all'utente di vedere il risultato calcolato dalla macchina in forma numerica (per esempio mediante un display numerico tipo quello di una calcolatrice tascabile), un numero alla volta.

RAM ("random access memory", traducibile in italiano con "memoria ad accesso arbitrario mediante indirizzo")
Realizza la memorizzazione di un vettore (sequenza con posizioni numerate) di numeri interi. Sia la dimensione del vettore (numero di elementi componenti) che il massimo valore memorizzabile in ogni elemento del vettore sono predeterminati al momento della costruzione e/o assemblaggio del dispositivo. Inteso come macchina virtuale, supporta l'esecuzione di due istruzioni fondamentali:

1. memorizza (scrive) il numero V nella cella numero N;

2. recupera (legge) il valore V precedentemente memorizzato nella cella numero N (mantenendolo comunque memorizzato, in modo da poter eseguire una sequenza arbitrariamente lunga di operazioni di lettura fornendo sempre l'ultimo valore memorizzato nella cella indicata dall'indirizzo N).

CU (unità di controllo)
realizza il funzionamento della macchina secondo le modalità spiegate nel seguito. Contiene due *registri* in grado di memorizzare valori numerici interi, esattamente come le celle della RAM: un registro chiamato "accumulatore" ed un "registro delle istruzioni" (Instruction Register, abbreviato IR). Contiene inoltre un terzo registro chiamato "contatore di programma" (Program Counter, abbreviato PC) in grado di contenere un valore numerico corrispondente ad un indirizzo per individuare una cella di RAM.

Per fissare le idee possiamo ipotizzare che l'unità RAM disponga di mille elementi (o *celle*, o *parole*) di memoria, e che ciascuna cella di memoria sia quindi univocamente individuata da un numero compreso tra 0 e 999. Potremmo quindi indicare con la notazione RAM[0] la prima cella, RAM[5] la sesta, e RAM[999] la millesima e ultima parola di memoria. Complessivamente la nostra macchina potrebbe quindi "ricordare" fino ad un massimo di 1000 numeri diversi. Sempre a titolo di esempio, potremmo fissare una volta per tutte che i numeri su cui vogliamo far operare il nostro sistema di calcolo siano i numeri interi positivi con rappresentazione decimale su 4 cifre, quindi ogni cella di RAM ed i registri accumulatore e IR potrebbero memorizzare valori compresi tra 0 e 9999. Il registro PC dell'unità di controllo potrebbe invece essere realizzato in modo piu' semplice, essendo richiesto che possa contenere valori rappresentabili con sole tre cifre decimali (gli indirizzi di RAM compresi tra 0 e 999).

L'unità di controllo deve essere realizzata in modo da poter eseguire (direttamente oppure comandando parte dell'esecuzione alle altre tre componenti della macchina) un insieme predefinito di istruzioni. Sempre a titolo esemplificativo potremmo individuare un insieme di nove istruzioni base. L'idea fondamentale di Von Neumann (risalente in realtà ai lavori più teorici dell'altro matematico inglese Alan Turing), che viene ancora oggi seguita nella realizzazione dei sistemi di calcolo fu quella di *codificare* le istruzioni in forma numerica e di inserirle, *insieme agli eventuali dati*, nella unità RAM della macchina. Un modo molto semplice per ottenere una codifica numerica per l'identificazione delle istruzioni è quello di elencarle in una lista ordinata e di usare il numero d'ordine all'interno della lista per individuare l'istruzione. Per chiarire il concetto proviamo a considerare il seguente esempio:

istruzione 0 (con parametro N): somma di due numeri
somma al valore memorizzato nell'accumulatore il valore memorizzato nella cella di memoria numero N e memorizza il risultato nell'accumulatore al posto del valore iniziale.

istruzione 1 (con parametro N): sottrazione tra due numeri
sottrai al valore memorizzato nell'accumulatore il valore memorizzato nella cella di memoria numero N e memorizza il risultato nell'accumulatore al posto del valore iniziale. Poichè la macchina non sa rappresentare valori negativi, si stabilisce per convenzione che se il valore contenuto nella cella di memoria è maggiore di quello memorizzato inizialmente nell'accumulatore, il risultato inserito nell'accumulatore alla fine dell'esecuzione dell'istruzione vale zero.

istruzione 2 : ingresso di un numero
si attende che l'utente inserisca un valore sul dispositivo di ingresso, dopo di che tale valore viene memorizzato nel registro accumulatore (al posto del valore precedente).

istruzione 3 : uscita di un numero
il valore memorizzato nel registro accumulatore viene trasferito per la stampa sul dispositivo di uscita (senza perdere memoria del valore nel registro accumulatore).

istruzione 4 (con parametro N): scrittura in memoria di un numero
memorizza il valore contenuto nel registro accumulatore anche all'interno della cella RAM[N] (senza perdita di memoria da parte del registro accumulatore, ma con cancellazione del valore precedentemente memorizzato in RAM[N]).

istruzione 5 (con parametro N): lettura dalla memoria di un numero
copia il valore precedentemente memorizzato nella cella RAM[N] anche nel registro accumulatore; il registro accumulatore perde traccia del valore in esso precedentemente memorizzato, mentre la cella RAM[N] mantiene invariata la sua memoria.

istruzione 6 (con parametro N): salto incondizionato di programma
memorizza il valore numerico N nel registro PC, perdendo quindi memoria del valore precedente nel contatore di programma;

istruzione 7 (con parametro N): salto condizionale di programma
controlla il valore numerico memorizzato nel registro accumulatore; se il valore è zero allora esegui le stesse operazioni definite per l'istruzione 6, altrimenti non fare niente.

istruzione 8 : arresto
termina l'interpretazione del programma non eseguendo più alcuna istruzione dopo questa.

Tralasciando per il momento le istruzioni 6, 7 e 8, possiamo fare tre considerazioni essenziali:

1. ogni istruzione può essere codificata mediante una singola cifra decimale compresa tra 0 e 8, indipendentemente dalla complessità di realizzazione della istruzione stessa. Come si realizza la somma tra due interi rappresentati su 4 cifre decimali è irrilevante ai fini della codifica della prima istruzione con la cifra decimale "0". La tabella su riportata è sufficiente per effettuare la decodifica delle istruzioni ed attivarne l'esecuzione (a patto che la macchina sappia effettivamente cosa fare per realizzare l'istruzione).

2. alcune istruzioni come la "2" e la "3" sono univocamente determinate dalle cifre decimali che le rappresentano, mentre altre (la "0", la "1", la "4" e la "5") richiedono la definizione di un ulteriore parametro (il valore N dell'indirizzo della cella di memoria) per individuare il secondo operando. La codifica di istruzioni che prevedono l'individuazione esplicita di operandi (il registro accumulatore individua in modo implicito un operando su cui applicare l'operazione) richiede la rappresentazione esplicita anche di tali operandi. Quindi nel caso delle istruzioni "0", "1", "4" e "5" occorre codificare anche il valore N.

3. le istruzioni proposte permettono di effettuare semplici manipolazioni aritmetiche su dati. Abbiamo bisogno di poter comporre più istruzioni macchina per costruire dei *programmi* per la soluzione di problemi. Un modo molto semplice di comporre programmi consiste nello specificare *sequenze* di istruzioni elementari da eseguire una dopo l'altra. Per esempio, se vogliamo stampare la somma tra due numeri usando la nostra macchina, potremmo leggere il primo addendo (istruzione "2"), poi memorizzarlo nella cella RAM[50] (istruzione "4 con parametro 50"), poi leggere il secondo addendo (istruzione "2" di nuovo), poi sommare all'accumulatore il contenuto di RAM[50] (istruzione "0"), ed infine stampare il risultato (istruzione "3").

Supponendo di saper codificare qualsiasi istruzione (compresi i suoi eventuali operandi) sotto forma numerica in una singola cella di RAM, allora possiamo individuare una modalità di funzionamento *sequenziale* per la nostra macchina facendo uso dei registri PC ed IR, esprimibile con la definizione delle seguente sequenza di fasi:

reset: Inserisci il valore 0 nel registro PC.

fetch: Estrai dalla memoria il valore RAM[PC] e memorizzalo nel registro IR. Successivamente incrementa di 1 il valore memorizzato nel registro PC.

decode: Osserva il valore contenuto nel registro IR ed interpretalo come la rappresentazione di una istruzione da eseguire.

execute: Esegui l'istruzione decodificata al passo precedente. Una volta terminate le operazioni richieste, torna alla fase di "fetch", a meno che l'istruzione da eseguire non fosse quella di terminazione del programma (la "8" nel nostro esempio).

Applicando questo semplice algoritmo, l'unità di controllo è in grado di eseguire programmi sequenziali le cui istruzioni sono codificate sotto forma numerica nelle celle della RAM a partire dall'indirizzo 0. Ovvero, RAM[0] contiene la codifica della prima istruzione del programma, RAM[1] contiene la codifica della seconda istruzione del programma, ecc.

Il caricamento di programmi sequenziali nella RAM (codificati in forma numerica) costituisce quindi un mezzo semplice ma molto efficace per definire sequenze di istruzioni (anche molto lunghe) che la macchina dovrà eseguire una volta attivata (per esempio attraverso la pressione di un apposito pulsante di avvio).

Dopo aver visto l'implementazione della esecuzione sequenziale di un programma mediante l'algoritmo di fetch-decode-exec, possiamo quindi passare ad esaminare il significato e l'uso delle istruzioni di "controllo di flusso" esemplificate nella nostra MVN dalle istruzioni "6", "7" e "8". Tali istruzioni servono per alterare, quando necessario, l'esecuzione sequenziale di programmi codificati in memoria. L'istruzione "6" viene chiamata normalmente *salto di programma* in quanto determina una interruzione nella continuità della scansione della RAM in forma sequenziale: fa sì che la prossima istruzione da eseguire venga decodificata a partire dal contenuto della cella RAM[N] anzichè dal contenuto della cella di indirizzo successivo rispetto a quella che contiene la codifica dell'istruzione di salto. L'istruzione "7" viene chiamata *salto condizionale* perchè l'interruzione della sequenzialità del programma avviene solo se è verificata una particolare condizione (il fatto che

sia stato precedentemente memorizzato il valore 0 nell'accumulatore). Notare che anche le istruzioni "6" e "7" richiedono la codifica di un parametro numerico oltre all'individuazione della istruzione, mentre l'istruzione "8" di terminazione è priva di parametri.

Per completare il nostro esempio di MVN non ci resta che definire la codifica numerica delle istruzioni (comprese quelle che richiedono il parametro N). Un modo semplice per ottenere tale codifica consiste nel moltiplicare per 1000 la cifra decimale che rappresenta l'istruzione e sommare il valore dell'eventuale operando N (sempre compreso tra 0 e 999. a causa della limitazione sulla dimensione della RAM). La decodifica potrà essere effettuata prendendo la parte intera della divisione del valore numerico diviso 1000 per ottenere la cifra decimale che rappresenta l'istruzione, ed il resto di questa stessa divisione come valore dell'indirizzo della cella di memoria da usare come operando.

2.3.2 Un esempio di programmazione della MVN

Consideriamo l'esempio più semplice che ci possa venire in mente: la realizzazione della somma tra due numeri. Il problema può essere risolto introducendo la seguente sequenza di numeri nella RAM (a partire dall'indirizzo 0):

0	2000	// legge un valore da input nell'accumulatore
1	4000	// memorizza il contenuto dell'accumulatore in RAM[0]
2	2000	// legge un valore da input
3	0000	// somma all'accumulatore il valore presente in RAM[0]
4	3000	// scrive in output il valore dell'accumulatore
5	8000	// termina

Mettendoci nei panni dell'unità di controllo possiamo facilmente verificare che applicando le fasi di fetch-decode-exec a partire dal valore PC=0, questo programma legge il primo addendo, lo memorizza nella cella di indirizzo 0, legge il secondo addendo, lo somma al contenuto della cella RAM[0], manda il risultato in uscita e termina. Notare che la cella RAM[0] viene usata (in tempi successivi) prima per contenere il codice di una istruzione, e poi per memorizzare un dato su cui opera il programma. Questo non crea problemi di interpretazione da parte dell'unità di controllo perchè la corretta interpretazione del valore recuperato dalla cella RAM[0] dipende dalla fase in cui l'unità di controllo si trova: il valore viene interpretato come la rappresentazione di una istruzione da eseguire se viene recuperato nella fase di fetch, mentre viene interpretato come dato da manipolare se viene recuperato nella fase di esecuzione. Dal punto di vista della macchina virtuale, si potrebbe ottenere esattamente lo stesso risultato sostituendo i valori memorizzati nelle celle di indirizzo 1 e 3 rispettivamente con 4006 e 6. Questa variazione sul tema potrebbe risultare forse più facile da comprendere da parte di un umano, mentre dal punto di vista della macchina virtuale l'unica differenza sarebbe la necessità di usare 7 celle di memoria invece di 6.

Proviamo a introdurre una variante più significativa, definita come segue:

0	2000	// legge un valore da input nell'accumulatore
1	7009	// se l'accumulatore contiene il valore 0 allora mette 9 nel PC
2	4008	// memorizza il contenuto dell'accumulatore in RAM[8]
3	2000	// legge un valore da input
4	7009	// se l'accumulatore contiene il valore 0 allora mette 9 nel PC
5	0008	// somma all'accumulatore il valore presente in RAM[8]
6	3000	// scrive in output il valore dell'accumulatore
7	6000	// mette 0 nel PC
8	0000	// questa cella viene utilizzata solo per i dati
9	3000	// scrive in output il valore dell'accumulatore
10	8000	// termina

In questo caso il programma di somma può essere ripetuto per un numero arbitrario di volte, fino a quando uno dei due addendi non assume il valore 0. In questo caso è cruciale usare una cella separata (nel nostro esempio RAM[8]) per la memorizzazione del primo addendo, in quanto le celle contenenti la rappresentazione di istruzioni devono essere consultate (fetch) più volte.

La banalità del nostro primo esempio di programma deriva dal fatto che stiamo semplicemente sfruttando le capacità di elaborazione del sistema: l'operazione di somma richiesta non è altro che una delle istruzioni di base che la nostra unità di controllo "sa fare" a livello hardware. Proviamo invece a vedere un esempio meno banale come quello specificato di seguito:

0	2000	// legge un valore da input nell'accumulatore
1	7019	// se l'accumulatore contiene 0 allora salta all'istruzione 19
2	4015	// memorizza il contenuto dell'accumulatore in RAM[15]
3	4016	// memorizza il contenuto dell'accumulatore in RAM[16]
4	2000	// legge un valore da input nell'accumulatore
5	7019	// se l'accumulatore contiene 0 allora salta all'istruzione 19
6	1014	// sottrae dall'accumulatore il valore presente in RAM[14]
7	7018	// se l'accumulatore contiene 0 allora salta all'istruzione 18
8	4017	// memorizza il contenuto dell'accumulatore in RAM[17]
9	5016	// copia nell'accumulatore il valore in RAM[16]
10	0015	// somma all'accumulatore il valore presente in RAM[15]
11	4016	// memorizza il contenuto dell'accumulatore in RAM[16]
12	5017	// copia nell'accumulatore il valore in RAM[17]
13	6006	// salta all'istruzione 6
14	0001	// questa cella contiene un valore costante 1
15	0000	// questa cella contiene dati (valore di a)
16	0000	// questa cella contiene dati (prodotto parziale)
17	0000	// questa cella contiene dati (valore di $b - i$)
18	5016	// copia nell'accumulatore il valore in RAM[16]
19	3000	// scrive in output il valore dell'accumulatore
20	8000	// termina

Emulando il comportamento dell'unità di controllo possiamo vedere che in questo caso il programma legge un primo valore "a" dal dispositivo di ingresso, poi si chiede se questo è zero, e nel caso termina dopo aver stampato in uscita il valore 0. Altrimenti (se $a > 0$) legge un secondo valore "b" e ripete il confronto col valore zero come prima. Se anche il secondo valore letto è $b > 0$, allora esegue la sommatoria:

$$\sum_{i=1}^{b} a$$

portando quindi a terminare dopo aver visualizzato il prodotto $a \cdot b$. Quest'ultimo esempio ci dimostra la possibilità di usare una macchina programmabile per svolgere attività più complesse di quelle disponibili al solo livello hardware, grazie all'interpretazione di un opportuno programma che descrive come ottenere un risultato complesso (il prodotto, che la nostra unità di controllo da sola non "saprebbe fare") attraverso una sequenza di passi più semplici (le somme, sottrazioni e confronti con 0, che l'unità di controllo "sa fare").

2.3.3 Le macchine convenzionali moderne

Nel tentativo di offrire una macchina virtuale più facile da utilizzare rispetto ai primi prototipi tipo Von Neumann, i ricercatori hanno lavorato per molti anni alla ricerca della struttura fisica e dell'insieme di istruzioni "ottimali" per la realizzazione a livello hardware dei processori. Una parte di questa storia (per altro interessante) verrà spiegata in modo relativamente approfondito nel corso di Sistemi di Elaborazione dell'Informazione. Ci limitiamo qui ad osservare che, per ragioni di semplicità di realizzazione, ci si è concentrati sulla rappresentazione "in base 2" per i numeri (anzichè in base 10 come implicitamente assunto fino a questo punto delle dispense) e si è deciso di definire un insieme di istruzioni tale da consentire di realizzare qualsiasi programma senza dover ricorrere al "trucco" della automodifica. Quest'ultima caratteristica permette di utilizzare eventualmente anche dispositivi di memoria a sola lettura (Read-Only Memory, abbreviato ROM) per la codifica delle istruzioni.

Si vedranno in corsi successivi alcuni esempi (ancora molto semplificati rispetto alla realtà) di macchine convenzionali "molto più semplici da programmare" rispetto al nostro esempio di MVN. Bisogna notare tuttavia che, per quanti sforzi siano stati fatti, ci si è sempre scontrati con la difficoltà di realizzare fisicamente una macchina virtuale sufficientemente sofisticata da poter essere "programmata facilmente" da un umano. Un altro modo forse più preciso di descrivere il problema può essere quello di dire che, nonostante si sia provato a realizzare fisicamente macchine virtuali sempre più sofisticate, non si è mai riusciti ad ottenere macchine "veramente semplici" da programmare.

Sui libri classici di programmazione (tutti scritti in inglese) si trova spesso il termine di "semantic gap" (divario semantico) tra quello che la realizzazione di una macchina programmabile richiede e quello che viene considerato semplice e intuitivo da parte di una persona. Un modo abbastanza efficace di affrontare questo problema è quello di definire una serie di macchine virtuali a "livelli di astrazione diversi", con la possibilità di tradurre automaticamente il programma per una macchina virtuale di alto livello (linguaggio di programmazione, sufficientemente vicino al modo di pensare di un programmatore umano quando affronta un certo tipo di problemi) nel programma di una macchina virtuale di livello più basso (il codice eseguibile direttamente da una macchina fisica).

Gli informatici hanno sviluppato a partire dagli anni ’60 ad oggi diverse tecniche di programmazione che fanno normalmente uso di “linguaggi (di programmazione) ad alto livello” e (giustamente) si rifiutano ormai di programmare in “linguaggio macchina”. Il criterio principale che viene oggi usato per giudicare la “bontà” di una macchina convenzionale è quindi il grado di semplicità e di efficienza con cui questa macchina convenzionale consente di riprodurre i costrutti di programmazione presenti nei moderni linguaggi ad alto livello.

Parte 3

Esecuzione dei programmi

Le architetture di calcolo reali sono coerenti con la struttura descritta finora, ma hanno un'organizzazione indubbiamente molto più complessa. In questo capitolo, pur mantenendo un livello di astrazione piuttosto elevato, introdurremo diversi concetti che riguardano l'ambiente di esecuzione dei programmi (*Runtime environment*) e che sono cruciali per la comprensione di gran parte degli argomenti inerenti la programmazione in generale e la programmazione imperativa in particolare.

3.1 Generalità su linguaggi, compilatori, sistema operativo

Forniamo di seguito solo brevi cenni ad argomenti che saranno svolti in modo approfondito nell'ambito di altri corsi. Questi cenni sono necessari a capire sia il contesto in cui si muove il programmatore quando sviluppa i programmi, sia il contesto in cui si trovano i programmi stessi quando vengono eseguiti.

I programmi che "girano" su un sistema di calcolo sono sequenze di istruzioni e dati in codice macchina, simili a quelle viste nel capitolo sulla MVN (benché molto più complesse) e ascrivibili al livello L2 tra quelli elencati in Sezione 2.2. I linguaggi di alto livello (livello L5) forniscono un supporto logico che permette al programmatore di lavorare in modo simbolico, astraendo rispetto a moltissimi dettagli del codice macchina. Tale livello di astrazione risulta molto più vicino alla comprensione umana ma, di conseguenza, necessita di un processo di "traduzione" per rendere i programmi eseguibili sul sistema di calcolo.

Il *compilatore* è un programma che prende in input il codice di un programma scritto in un linguaggio di alto livello e lo traduce (si dice *compila*) in codice eseguibile. Questo processo chiama in causa un altro programma (che spesso non viene distinto dal compilatore stesso) noto come *linker*, che si occupa di "completare" il programma compilato, unendolo ad altri "pezzi" di codice pre-compilato, provenienti da librerie o da altri moduli scritti dallo stesso programmatore. Senza entrare in ulteriori dettagli, notiamo i seguenti fatti:

- Il compilatore analizza la *sintassi* del codice in input e lo traduce (con l'aiuto del linker) in codice eseguibile. Se l'operazione di compilazione ha successo, si ha la garanzia che il codice sia sintatticamente

corretto e che abbia generato un programma eseguibile ma *non si ha nessuna garanzia* sulla correttezza *semantica* del programma stesso, ossia che il programma svolga effettivamente e correttamente il compito per il quale è stato progettato. Anche il fatto che il programma si comporti nel modo voluto su uno o alcuni esempi di input non dà nessuna garanzia sulla sua correttezza. D'altra parte, dimostrare la correttezza semantica di un programma è un'operazione improba, fattibile solo in casi molto semplici e sostanzialmente improponibile nei casi complessi: in generale, è proprio impossibile dimostrare se un programma qualsiasi terminerà o meno in presenza di un input qualsiasi, figuriamoci dimostrare se terminerà producendo il risultato corretto. Pertanto, la correttezza semantica di un programma viene normalmente verificata attraverso un processo di *testing*, facendo girare il programma su molti input che rappresentino una casistica il più ricca possibile. Più il programma è complesso, più lo sarà il relativo processo di testing.

- Per quanto alto sia il livello di astrazione di un linguaggio, spesso risulta molto oneroso sviluppare un programma complesso da zero utilizzando i soli costrutti base del linguaggio stesso. Per questo motivo, la maggior parte dei programmi vengono sviluppati facendo ricorso a *librerie* che forniscono moduli precompilati, utilizzabili come estensioni procedurali, con funzioni di livello ancor più alto delle semplici istruzioni del linguaggio. La semantica delle funzioni di libreria viene specificata in modo più o meno informale (ma di solito sufficientemente comprensibile) nei manuali delle librerie stesse. I principi di *programmazione strutturata* che seguiremo nell'ambito di questo corso, suggeriscono inoltre anche un'organizzazione il più possibile modulare del software con un uso molto spinto delle estensioni procedurali e della compilazione separata. Quindi, si darà il caso molto spesso che un programma sviluppato da un unico programmatore sia in effetti costituito da più moduli compilabili separatamente e "cuciti insieme" dal linker.

- In ogni caso, in questo corso ci limiteremo a trattare programmi *sequenziali* (cioè in cui viene eseguita una istruzione alla volta e le varie istruzioni vengono eseguite in sequenza) costituiti da un programma principale (detto funzione `main` in linguaggi come C e C++) più eventuali estensioni procedurali (funzioni e procedure). Nella (buona) pratica, le estensioni procedurali di solito costituiscono la maggior parte del codice di un programma strutturato e possono provenire in parte dal codice scritto dal programmatore e in parte dalle librerie.

Come già accennato, il *sistema operativo* è quel programma che viene caricato dal bootstrap e che continua a girare e a presiedere all'attività degli altri programmi finché il sistema è in funzione. Il sistema operativo può disporre, secondo le proprie esigenze o quelle dell'utente, l'esecuzione contemporanea di più programmi, secondo il già accennato schema di time sharing. Ciascuno di questi programmi in esecuzione viene detto *processo* e il sistema operativo assegna a ciascun processo una porzione di memoria RAM, organizzata al proprio interno secondo uno schema simile a quello presentato nel Capitolo 4, che sarà dettagliato in una delle prossime sezioni. Per i nostri scopi, possiamo assumere di trattare un solo programma alla volta e che la memoria a disposizione del corrispondente processo sia totalmente separata da quella a disposizione di altri processi.

3.2 Rappresentazione dei dati: tipi

Benché a basso livello qualunque informazione gestita da un programma sia codificata da sequenze di bit, nei linguaggi di alto livello si può usare (per fortuna) un livello di astrazione molto superiore, classificando l'informazione in molti *tipi*, alcuni dei quali sono predefiniti, mentre altri possono essere definiti dal programmatore. L'idea generale è che il tipo relativo ad una data sequenza di bit determina la "comprensione" del suo significato da parte del sistema stesso, dove "comprensione" va intenso in senso lato: il sistema tratterà i bit della sequenza coerentemente con le regole di manipolazione previste per quel tipo. Il tipo è pertanto fondamentale affinché il programma non faccia confusione: il programma "sa" come interpretare e manipolare un dato secondo il tipo del dato stesso.

Prima di tutto, un tipo caratterizza un *dominio* di valori modificabili e anche lo spazio necessario (in bit) per codificare uno di questi valori. A titolo di esempio, finora abbiamo incontrato dati di tre tipi: numeri interi senza segno, numeri relativi (interi con segno) e numeri razionali (che nei sistemi di calcolo si usano come approssimazione dei numeri reali). Molti linguaggi di programmazione (come ad esempio C, C++, Pascal , ecc.) forniscono schemi e nomi predefiniti per rappresentare questi tipi. In C e C++, per esempio: il tipo `int` denota gli interi con segno, il tipo `unsigned` i naturali, i tipi `float` e `double` i razionali floating point (la differenza tra gli ultimi due tipi sta nel numero di bit usati per la codifica, che è doppio nei double rispetto ai float). Esistono parecchie altre varianti, ad esempio interi rappresentati su più o meno bit, che per il momento possiamo tralasciare. Altri tipi notevoli di uso comune sono:

- Booleani (`bool` in C++, non presente in C): è il tipo usato nell'algebra di Boole, corrispondente al dominio che contiene i due soli valori *vero* e *falso*. Un valore di questo tipo è rappresentabile su un singolo bit, anche se spesso si usa almeno un byte (8 bit), con conseguente spreco di spazio, perché indirizzare singoli bit nella memoria può essere dispendioso in termini di tempo. La rappresentazione a singolo bit può essere utilizzata in pratica quando si trattano "pacchetti" di diversi Booleani che possono essere racchiusi otto a otto in "scatole" grandi un byte. Di solito, ma non necessariamente, si usa la convenzione che 0 significhi falso e 1 significhi vero: questa convenzione viene usata implicitamente anche in C, dove il tipo non è presente. I Booleani svolgono un ruolo primario nella programmazione, specialmente - ma non solo - nella gestione delle strutture di controllo come istruzioni condizionali e cicli. Nel capitolo sulla MVN, abbiamo già incontrato il salto condizionale che rappresenta il prototipo, a basso livello, del controllo di flusso del programma. In quel caso, il salto è condizionato dal valore Booleano generato dalla risposta alla domanda: "l'accumulatore contiene il valore zero?".

- Caratteri (`char` in C e C++): tipo che denota l'insieme di tutti i caratteri alfanumerici, tra i quali quelli presenti sulla tastiera, ma anche altri che sono noti come caratteri di controllo (non stampabili), come ad esempio il carattere che segna la fine di una linea di testo ed il passaggio alla successiva. I caratteri vengono reppresentati internamente da codici numerici tramite una tabella di conversione: ad ogni carattere viene associato un numero naturale. La tabella base è la ASCII, che prevede codici a 7 bit ed è quindi in grado di rappresentare solo 128 caratteri. Esistono poi tabelle più estese, tra cui gli standard Unicode UTF-8, che è compatibile con ASCII e la estende su 8 bit, e UTF-16 a 16 bit.

- Puntatori: si tratta di indirizzi di altri dati. La differenza rispetto agli indirizzi di basso livello della RAM è abbastanza sottile e può essere spiegata come segue: un puntatore non indirizza semplicemente una cella di memoria RAM, ma indirizza l'intero spazio occupato da un dato di un certo *tipo*; quindi il puntatore normalmente porta in sé sia la conoscenza di dove è situata l'area di memoria contenente il dato, che di quanto questa area sia estesa e della natura (tipo) del dato che contiene.
- Stringhe: si tratta di sequenze di caratteri, di lunghezza arbitraria. Molto spesso le stringhe non vengono fornite come tipi elementari nei linguaggi di programmazione, ma gestite mediante *costruttori di tipi* più generali, come gli *array* (vedi sotto). In C e C++ sono presenti librerie standard che non fanno parte del linguaggio in senso stretto, ma che vengono utilizzate comunemente come se ne facessero parte. Tali librerie forniscono questo tipo ed un supporto piuttosto esteso al trattamento delle stringhe.

Gli *array*, gli *stream* e i *record* (`struct` o `class` in C e C++) sono *costruttori di tipo* che servono ad organizzare dati omogenei (array e stream) o eterogenei (record) in "contenitori" che definiscono tipi *strutturati*, comunemente definiti dal programmatore. Le librerie standard spesso forniscono altri contenitori e dati strutturati, costruiti mediante questi strumenti, che possono essere anche piuttosto complessi e possono risparmiare molta fatica in diverse situazioni di uso comune. Per il momento non sviluppiamo ulteriormente questi concetti, che saranno poi studiati nel contesto della programmazione imperativa.

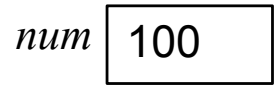
3.3 Rappresentazione dei dati: variabili

Nella programmazione delle macchine convenzionali vista fino al capitolo precedente, i dati erano contenuti in locazioni di memoria o nei registri. Gli indirizzi di tali locazioni e i nomi dei registri erano gli unici strumenti a disposizione del programma per accedere ai dati stessi. Nonostante questo resti comunque il modo effettivo che il sistema usa per accedere ai dati, nei linguaggi di più alto livello è possibile associare "nomi" ai dati, in modo da permettere al programmatore una più facile scrittura e comprensione del codice. I nomi definiti nel contesto di un programma si chiamano *identificatori* e un dato associato ad un identificatore si chiama *variabile*. Una *dichiarazione di variabile* consiste di un pezzo di codice che *realizza l'associazione logica tra un identificatore ed un'area di memoria atta a contenere un dato di un certo tipo.* Ad esempio, in C o C++ la dichiarazione

```
int num;
```

sta dicendo al programma: "riserva nella tua memoria lo spazio per un dato di tipo intero con segno e sappi che da ora in poi tutte le volte che troverai l'identificatore `num` nel codice, questo si riferisce a quel dato" (n.b.: vedremo più avanti che "da ora in poi" va interpretato secondo regole abbastanza delicate: in realtà ogni identificatore possiede un ambito nel quale "vive" che non corrisponde necessariamente a tutto il programma dal punto della sua dichiarazione in poi).

Nel seguito del corso useremo il formalismo grafico seguente: una variabile sarà denotata da un rettangolo; a lato di questo rettangolo scriveremo il nome della variabile, mentre dentro il rettangolo scriveremo il valore corrente della variabile. Nell'esempio seguente, il valore corrente della variabile intera *num* è 100.



Il nome serve al programma per rintracciare la variabile (cioè funziona da indirizzo simbolico) mentre tutte le operazioni di lettura e scrittura riguardano il valore contenuto nel rettangolo. È di fondamentale importanza ricordare due cose:

1. L'identificatore associato ad una variabile può denotare sia il *contenitore* (cioè l'area di memoria utilizzata per mantenere la variabile) che il suo *contenuto* (cioè il valore corrente di quella variabile, interpretato secondo il suo tipo). Per una ragione che sarà chiarita in seguito - e che renderà semplice memorizzare la terminologia - si parla di *valore sinistro* della variabile per intendere il suo contenitore e *valore destro* per intendere il suo contenuto. Come nella MVN, le operazioni che si compiono sui dati sono essenzialmente solo due: si può *scrivere* nel contenitore oppure si può *leggere* il contenuto. Nei linguaggi di alto livello è sempre evidente dal contesto se una variabile viene trattata come valore sinistro o valore destro.

2. In una variabile è *sempre* presente un valore. In generale, una dichiarazione si limita a riservare il contenitore, lasciando il valore indefinito. Indefinito **non** significa che non c'è ma solo che potrebbe essere qualunque valore. È compito dell'*inizializzazione*, che è un'istruzione nel codice del programma, inserire un primo valore nella variabile. Linguaggi come C e C++ forniscono strumenti per racchiudere dichiarazione ed inizializzazione nell'ambito della stessa istruzione: ad esempio

   ```
   int num = 20;
   ```

   non solo dichiara la variabile intera `num` ma le assegna anche il valore 20. L'inizializzazione contestuale alla dichiarazione non è obbligatoria, ma bisogna sempre ricordare che se si accede a una variabile in lettura prima di averla inizializzata, il contenuto della variabile è un valore arbitrario, quindi in generale non valido ai fini del programma. Quindi, per garantire il corretto funzionamento del programma, una variabile deve sempre essere *scritta* almeno una volta *prima* di essere *letta*.

   Attenzione: La mancata inizializzazione è una delle fonti più comuni di errore e può essere molto subdola: un programma contenente una mancata inizializzazione può girare correttamente se compilato con un dato compilatore (che ad esempio inizializzi automaticamente le variabili all'atto della dichiarazione in un modo che, solo per fortuna, è compatibile col funzionamento del programma stesso) e piantarsi sistematicamente se compilato con un altro compilatore (che non inizializza affatto le variabili o le inizializza in modo incompatibile con gli scopi del programma); ancor peggio, dato che la mancata inizializzazione può lasciare la variabile "sporca" con valori casuali differenti da una volta all'altra, è possibile che lo stesso programma compilato una volta e fatto girare sempre sulla stessa macchina a volte giri correttamente e altre volte si pianti e pure che lo faccia in punti diversi: questi sono tra gli errori più difficili da individuare, benché conseguenti ad una dimenticanza molto banale. I moderni compilatori di solito segnalano queste dimenticanze mediante messaggi di *warning* che vengono visualizzati durante la fase di compilazione. Al contrario degli *errori* di compilazione, che impediscono al compilatore di arrivare in fondo al proprio compito e quindi devono essere obbligatoriamente risolti per poter compilare, gli

warning sono solo avvertimenti su possibili fonti di malfunzionamento del programma, che non ne impediscono la compilazione e l'esecuzione. In quanto tali, spesso vengono ignorati dai neofiti (e non) con conseguenze che a volte possono essere disastrose. Va detto che alcuni warning sono del tutto innocui e dovuti ad un eccesso di pedanteria del compilatore, anche in questo va cercata la tendenza di molti programmatori ad ignorarli; gli warning che avvertono della mancata inizializzazione di una variabile dovrebbero però essere presi sempre molto sul serio.

3.4 Rappresentazione dei dati: costanti

Per i tipi primitivi, i linguaggi forniscono comunemente simboli per denotare direttamente valori durante la scrittura dei programmi. In linguaggi come C, C++ e Pascal è possibile scrivere direttamente numeri interi (con o senza segno, come `234` oppure `-181`) o decimali (ad esempio in notazione col punto, come `-287.456`), caratteri che sono denotati tra apici (ad esempio `'a'` che denota la lettera a), stringhe denotate tra doppi apici (come `"pippo"`), Booleani (`true` e `false`), ecc.

Per comodità, in alcuni linguaggi è inoltre possibile associare nomi a valori costanti. Ad esempio in C++ la dichiarazione

```
const int DIM = 3;
```

sta dicendo al programma: "da ora in poi tutte le volte che trovi nel codice la parola DIM interpretala come il numero intero 3". Notiamo la profonda differenza tra una dichiarazione di variabile e una dichiarazione di costante, nonostante la sintassi simile. Anche quando sono definite da simboli, come in quest'ultimo caso, le costanti - in quanto tali - hanno un valore fissato, quindi non sono "dati" veri e propri: sono valori scritti nel codice che non possono variare né nel contesto di un'esecuzione né da un'esecuzione all'altra. Non occupano nessuna locazione nella memoria dedicata ai dati del programma ma risiedono nella parte dedicata al codice e il loro valore viene specificato nel codice stesso ad ogni occorrenza.

3.5 Organizzazione della memoria

Come anticipato in Sezione 3.1, assumiamo che un dato programma (incluse le sue estensioni procedurali) abbia a disposizione un dato segmento di memoria durante la sua esecuzione (si dice "a *run time*"). L'organizzazione di questa memoria raffina il modello a stack , organizzando la parte dedicata ai dati in tre sotto-segmenti, come descritto nello schema seguente. Per capire la differenza tra *stack, heap* e segmento per i dati globali e statici, anticipiamo alcune nozioni sulla struttura dei programmi e le regole di scope, che saranno poi sviluppate in seguito.

Un programma può essere visto come un insieme di scatole annidate, come schematizzato in Figura 3.2. I possibili schemi di annidamento delle scatole variano secondo i linguaggi di programmazione. Nel seguito as-

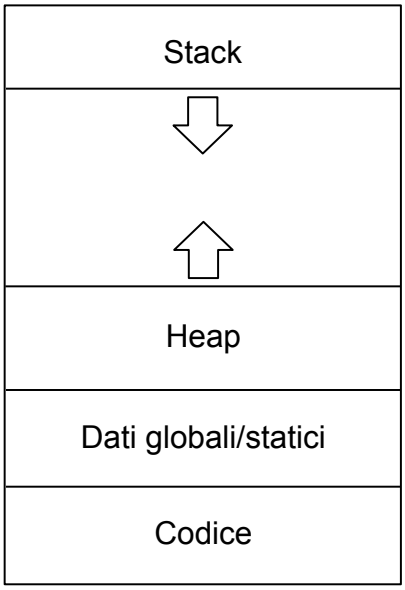


Figura 3.1: L'organizzazione della memoria a run time.

sumeremo lo schema usato nel linguaggio C (e nel C++, almeno nella sua forma più semplice di uso imperativo, prescindendo dalle sue caratteristiche object-oriented).

La scatola più esterna, corrisponde all'intero programma e si chiama *ambito globale*. Al suo interno sono presenti altre scatole, contenenti le varie estensioni procedurali (dette *funzioni* nei linguaggi C e C++), inclusa la funzione `main` che costituisce il programma principale. All'interno di ogni funzione possono essere annidati blocchi di codice. Il concetto di *blocco* è molto generico: denota un insieme contiguo di istruzioni racchiuse tra delimitatori che definiscono appunto i limiti della "scatola". In C e C++ un blocco è delimitato da una coppia di parentesi graffe { e }.

Le scatole (ossia: l'ambito globale, le funzioni e i blocchi) delimitano sia unità logiche di calcolo che ambiti di uso e "vita" delle variabili, governati da precise regole dette *regole di scope* (*scope* = ambito). Per i nostri scopi in questa sezione, tralasceremo tutto quanto riguarda il codice e ci occuperemo solo dei dati. Anche se nel seguito parleremo sempre di variabili, le stesse regole di scope valgono anche per le costanti, i tipi e in generale tutto quanto sia definito da identificatori dichiarati nel programma.

Presentiamo alcune regole di scope generali, semplificate assumendo che in un programma non compaiano mai due variabili diverse con lo stesso nome. Più avanti vedremo che questo non è affatto obbligatorio e che le regole di scope reali possono essere adattate facilmente a gestire la presenza di variabili con lo stesso nome, purché in ambiti diversi. Si parla di *visibilità* (o *scope*) di una variabile riferendosi alla possibilità di utilizzare tale variabile in un dato ambito; si parla di *vita* di una variabile riferendosi al lasso di tempo (a run time) nel quale tale variabile risulta disponibile.

1. *Variabili globali:* sono quelle dichiarate nell'ambito globale, vivono per tutta la durata del programma e sono visibili ovunque nel codice del programma.

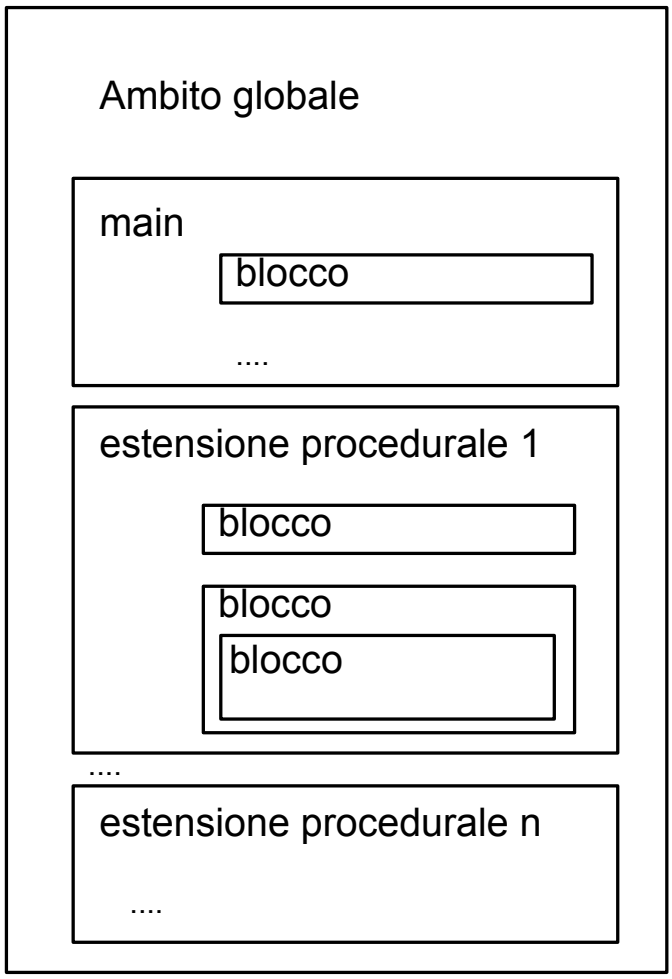


Figura 3.2: La struttura dei programmi C / C++.

2. *Variabili locali o automatiche:* costituiscono la maggior parte delle variabili dichiarate in un ambito (scatola) diverso da quello globale. Tali variabili sono visibili nell'ambito in cui sono dichiarate e in tutti gli ambiti annidati in esso, dal punto della dichiarazione di variabile in poi. Tali variabili vivono solo nel tempo di esecuzione del codice presente nell'ambito in cui sono dichiarate, inclusa l'esecuzione di tutti gli ambiti annidati.

   Attenzione: la funzione `main` non fa eccezione, le variabili locali ad essa *non* sono globali; tuttavia vivono per l'intera durata del programma perché la funzione `main` è la prima a partire e l'ultima a terminare.

3. *Variabili statiche:* sono dichiarate in un ambito non globale e sono visibili secondo le stesse regole di cui al punto 2, ma vivono per l'intera durata del programma secondo la regola di cui al punto 1. L'uso di queste variabili non è molto comune ed ha lo scopo di permettere ad un'estensione procedurale di essere eseguita più volte mantenendo memoria da una volta all'altra di dati locali ad essa ma salvati la volta precedente. Non faremo ulteriore uso di questa classe di variabili nel seguito di questo corso.

4. *Variabili dinamiche:* non sono presenti nel codice ma vengono create dal programma durante l'esecuzione. Di norma, tali variabili sono raggiungibili tramite puntatori, partendo di solito da variabili di tipo puntatore del programma. Le variabili dinamiche sono visibili ovunque siano visibili i puntatori che fanno riferimento ad esse e vivono per l'intero lasso di tempo che intercorre tra la loro creazione e la loro distruzione (o la fine del programma, se non vengono esplicitamente distrutte). Creazione e distruzione delle variabili dinamiche avvengono su esplicita richiesta del codice del programma mediante istruzioni apposite. *L'uso delle variabili dinamiche è molto diffuso e va trattato con molta attenzione.*

A questo punto possiamo descrivere l'organizzazione della memoria illustrata in Figura 3.1 e illustrarne a grandi linee la gestione.

- *Codice:* questa parte di memoria è occupata dalle istruzioni del programma. La parte di codice è statica, nel senso che le istruzioni vengono caricate in memoria al lancio del programma, vengono lette in sequenza (durante la fase di *fetch* di funzionamento della macchina - cfr. Sezione 2.3.1) ma non possono essere modificate dal programma.

- *Dati globali/statici:* tutte le variabili globali e le variabili statiche risiedono in un segmento di memoria la cui allocazione è statica - nel senso che avviene una sola volta al lancio del programma - mentre l'accesso è dinamico - nel senso che questa memoria può essere sia letta che modificata da qualunque istruzione del programma.

- *Stack:* Su di esso vengono allocate le variabili locali all'atto dell'attivazione del codice della corrispondente funzione o blocco. Il trattamento di funzioni e blocchi tuttavia è differente: mentre per le funzioni il meccanismo è del tutto coerente con quello descritto in precedenza, per i blocchi viene spostato solo il registro SP (per estendere la memoria con le variabili locali al blocco) ma non il registro FP (per permettere al blocco di mantenere la visibilità alle variabili provenienti dai blocchi in cui esso è annidato). Il meccanismo di gestione dello stack è piuttosto semplice, perché le regole per allocare e deallocare dinamicamente lo spazio necessario in ogni funzione o blocco vengono decise una volta per tutte dal compilatore (si dice a *compile time*, che è una fase off-line precedente al *run time*).

- *Heap:* lo heap è dedicato a mantenere le variabili dinamiche. Può essere considerato come un "serbatoio" al quale il programma può chiedere spazio a run time per allocare variabili che vengono create dal programma stesso e per le quali non era prevista l'allocazione a compile time. La gestione dello heap è molto più complicata di quella dello stack ed è demandata ad una funzione del sistema operativo che fa parte del runtime environment. Dal punto di vista del programmatore, l'interazione del programma col gestore dello heap consiste di due sole operazioni:

  – Richiesta spazio: il programma chiede al gestore di fornire lo spazio necessario per allocare un nuovo dato, specificando quanto spazio gli serve: ne servirà tanto quanto è previsto dal *tipo* del dato da allocare; il gestore dello heap individua tale spazio, eventualmente estendendo la propria area, e restituisce al programma un puntatore con il quale accedervi.

  – Restituzione di spazio: il programma dice al gestore che un dato spazio non gli serve più. Dal punto di vista del programma, significa che le variabili precedentemente allocate in tale spazio vengono

distrutte e non saranno mai più disponibili; dal punto di vista del gestore, significa che quello spazio potrà essere riutilizzato per soddisfare richieste successive da parte del programma.

Mentre la porzione di stack occupata dai dati è sempre costituita di un unico segmento di celle contigue, è facile immaginare come, alternando richieste a restituzioni, lo spazio a disposizione dello heap possa facilmente essere frammentato, rendendolo rapidamente un "groviera" di celle occupate e celle disponibili. Il gestore dello heap ha quindi un compito piuttosto complesso di amministrazione dello spazio: ad esempio, dovrebbe cercare di "riempire i buchi" prima di estendere la propria area, ma questo non è possibile quando gli arrivano richieste di segmenti più grandi dei "buchi" disponibili. La gestione dello heap può quindi risultare molto più costosa, in termini di risorse di calcolo e quindi di tempo impiegato, rispetto alla gestione dello stack. Lavorare con strutture dinamiche può essere efficiente dal punto di vista dello spazio occupato (solo lo stretto necessario in ogni momento), ma potrebbe risultare inefficiente dal punto di vista del tempo di calcolo, se la politica di allocazione e deallocazione del programma non è oculata.

Poiché sia lo stack che lo heap hanno bisogno di occupare una porzione di memoria che varia dinamicamente a run time, questi concorrono ad utilizzare lo stesso segmento di memoria: lo stack incomincia a occuparlo dagli indirizzi più bassi e cresce verso l'alto; viceversa, lo heap incomincia ad occuparlo dagli indirizzi più alti e decresce verso il basso. Entrambi hanno un comportamento "a fisarmonica", crescendo e decrescendo secondo gli eventi di allocazione e deallocazione. Tali eventi sono: esplicite richieste del codice per lo heap; ed implicite conseguenze dell'attivazione e terminazione di funzioni e blocchi per lo stack. L'invariante che deve essere sempre soddisfatto è che la somma dello spazio occupato da stack e heap (inclusi i suoi "buchi") non può mai superare la dimensione del segmento ad essi dedicato. Se le richieste eccedono la capacità disponibile il programma abortisce.

Parte 4

Stack di sistema

Le macchine convenzionali moderne offrono meccanismi ottimizzati per la soluzione del problema di attivazione di un sottoprogramma, passaggio dei parametri dal programma principale verso il sottoprogramma, allocazione di memoria "privata" per la procedura in modo da non interferire con la memoria usata dal programma principale, ritorno dei risultati dal sottoprogramma al programma principale e riattivazione del programma principale dal punto in cui era stato interrotto. Un modo semplice, efficace ed elegante per risolvere tutti questi problemi è l'adozione di una "struttura dati" chiamata *stack* da parte della macchina convenzionale.

Altra caratteristica che vale la pena richiamare dei moderni sistemi di calcolo è il fatto che *costringono ad una separazione netta del codice dai dati e favoriscono un uso ordinato e razionale dello spazio degli indirizzi in RAM*. In questo capitolo ci concentreremo sull'esistenza di un dato programma P1 (al quale verranno associate celle per il codice, distinte dalle celle per i dati) e di un secondo programma P2 destinato a risolvere un altro problema oppure a costituire una *estensione procedurale* (o funzione) utilizzata da P1. Anche P2 avrà una porzione di celle di memoria dedicate al codice e distinte da quelle dedicate ai dati.

In particolare, cerchiamo ora di risolvere il problema di consentire ai due programmi diversi P1 e P2 di accedere in modo semplice e uniforme ciascuno ai propri dati, indipendente dalla presenza di dati relativi ad altri programmi sempre allocati in RAM.

Intuizione. Il modello di uso dello spazio di memoria a cui facciamo riferimento è quello della sovrapposizione di nuovi fogli di carta (o quaderni) su di una scrivania. Supponiamo di essere seduti ad una scrivania e di studiare su una copia cartacea di questi appunti. A un certo punto durante la lettura incontriamo un termine inglese che non conosciamo. Prendiamo quindi il dizionario di inglese, che verrà posato sopra gli appunti, nascondendoli, e ci dedichiamo alla ricerca della traduzione del termine. Dopo aver trovato e tradotto il termine cercato, possiamo mettere via il dizionario e dedicarci nuovamente al nostro compito primario. Notiamo come la ricerca del termine nel dizionario assuma in questo contesto il ruolo di estensione procedurale del compito principale (studiare sugli appunti) e ne risolva una piccola parte (capire il significato di un termine ignoto). Notiamo anche come le informazioni necessarie alla soluzione del compito primario (gli appunti) siano solo *nascoste* durante l'esecuzione dell'estensione procedurale, ma non vadano mai perse e possano essere ripristinate quando lo svolgimento dell'estensione procedurale è terminato. Il termine ignoto da cercare e la sua

traduzione altro non sono che i parametri, rispettivamente di input e output, dell'estensione procedurale che vengono gestiti nella nostra memoria (che si presta a costituire una zona comune a entrambe le procedure) e vengono scambiati tra il programma primario e l'estensione stessa.

Notiamo infine che questo meccanismo non è necessariamente legato al fatto che un programma chiami una sua estensione procedurale (o funzione). In generale, un compito potrebbe essere interrotto per svolgere operazioni completamente estranee ad esso. Se durante lo studio arriva un amico e deposita un giornale sopra i fogli dei nostri appunti chiedendoci di dare un'occhiata alla pagina degli spettacoli, per decidere dove andare a teatro l'indomani sera, possiamo interrompere il nostro studio e dedicare alcuni minuti al problema della scelta dello spettacolo. Riprenderemo lo studio quando il giornale sarà ritirato e gli appunti "riemergeranno" alla superficie. Meccanismi simili vengono utilizzati per permettere ai sistemi di calcolo di eseguire più programmi nello stesso tempo. La contemporaneità dell'esecuzione è in effetti solo illusoria (o, al meglio, limitata al numero di processori che un dato sistema può utilizzare) mentre è in atto un meccanismo di *time sharing*: diversi programmi competono all'uso delle risorse di calcolo e il sistema operativo si occupa di attivarli ed interromperli ad intervalli, garantendo che ciascuno possa sempre accedere ai propri dati.

Indirizzamento indicizzato. Questo stesso modello di utilizzazione dello spazio di tipo *stack* può essere realizzato per l'accesso alle celle di RAM da parte di un programma mediante la definizione di un opportuno registro nella CPU (comunemente chiamato FP - dalle iniziali di *frame pointer* - oppure BP - dalle iniziali di *base pointer*) da utilizzarsi in abbinamento al *modo di indirizzamento indicizzato*. Per *modo di indirizzamento* si intende il meccanismo usato per calcolare l'indirizzo della cella di memoria da utilizzare come operando per l'esecuzione di una istruzione. Il modo di indirizzamento usato nella MVN (esempio: RAM[numero-costante]) prende il nome di *indirizzamento diretto*, perché l'indirizzo su cui operare viene specificato esplicitamente nel codice. Il modo indicizzato, viceversa, prevede di calcolare l'indirizzo della cella di memoria sommando un valore dato (costante, specificato nel codice) al contenuto di un registro (esempio: RAM[FP+5]).

Lo stack. Cominciamo quindi con l'identificare una zona di memoria RAM da dedicare alla realizzazione dello stack di sistema, per esempio le prime celle a partire dall'indirizzo 0 fino ad un certo valore, opportunamente elevato da garantire la disponibilità delle celle di memoria necessarie per qualsiasi programma volessimo eseguire (al solito, il termine *qualsiasi* va inteso in senso relativo: per quanto grande sia lo spazio predestinato ai dati, potremo sempre trovare un programma che ha bisogno di più memoria di quella disponibile; in quel caso, il programma sarebbe semplicemente troppo complesso per girare sulla macchina che abbiamo a disposizione). Ciascun programma farà uso di celle di memoria *contigue* in termini di indirizzo, che potranno essere visualizzate come "una fetta" della zona di RAM dedicata allo stack. Queste celle di memoria saranno dedicate *esclusivamente* a contenere i dati del nostro programma, pertanto né le istruzioni del nostro programma né, tanto meno, istruzioni e dati di altri programmi potranno occupare quella porzione di memoria fintantoché il nostro programma sarà in esecuzione.

Per esempio, supponiamo che il programma P1 necessiti di 10 celle di RAM per memorizzare i suoi dati, e che il programma P2 ne richieda invece 15. Ebbene, P1 individuerà le sue 10 celle di RAM mediante l'indirizzamento indicizzato: RAM[FP+0], RAM[FP+1], ..., RAM[FP+9]; P2 dal canto suo individuerà le sue 15 celle di RAM mediante l'indirizzamento indicizzato: RAM[FP+0], RAM[FP+1], ..., RAM[FP+14]. Sarà compito del "sistema operativo" (il programma caricato dal Bootstrap al momento dell'accensione e destinato a gestire le risorse del sistema) utilizzato su quella macchina convenzionale di assegnare preventivamente valori

diversi al registro FP per consentire a P1 di accedere alle "sue" dieci celle, separate e distinte dalle quindici celle assegnate a P2.

Supponiamo ora che il nostro ipotetico programma ad un certo punto dell'esecuzione richieda la sospensione di P1 e l'attivazione di P2 come estensione procedurale di P1, per realizzare una funzionalità complessa, non immediatamente riducibile alla esecuzione di una singola istruzione della macchina convenzionale. Al termine di P2 vorremo ovviamente che la macchina convenzionale riprenda l'esecuzione di P1 dal punto in cui era stata interrotta per consentire l'inizio della estensione procedurale P2.

Dal punto di vista dell'accesso ai dati, questa sostituzione può essere realizzata facendo in modo che nello stack l'area dati di P2 sia allocata "immediatamente dopo" quella di P1. Nel nostro esempio tale allocazione contigua avviene sommando al valore precedentemente usato da P1 per FP la costante 10 per ottenere il valore di FP che dovrà essere usato da P2 (in modo che FP+0 calcolato da P2 si riduca allo stesso valore che sarebbe stato calcolato sommando 10 al valore di FP durante l'esecuzione di P1). Per semplificare la realizzazione di questa operazione normalmente viene introdotto un secondo registro nella CPU (chiamato SP, abbreviazione di *stack pointer*) che delimita l'area dello stack usata dal programma in esecuzione, per esempio contenendo l'indirizzo della prima cella non usata subito dopo l'ultima cella usata dal programma.

Per tornare al nostro esempio, il programma P1 potrebbe essere mandato in esecuzione dopo aver prefissato il valore 0 nel registro FP (in modo da allocare le sue 10 celle di memoria dedicate ai dati nelle prime celle disponibili nello stack). Il programma stesso potrebbe calcolarsi il valore da assegnare al registro SP sommando 10 al valore contenuto nel registro FP. Notiamo che in questo caso 10 rappresenta il numero di celle di cui P1 ha bisogno ed è quindi un'informazione che P1 porta con sé (nel senso che deve essere scritta nel suo codice), mentre il valore di FP viene assegnato (dal sistema operativo, ovvero dal programma chiamante) quando P1 viene mandato in esecuzione e può variare di volta in volta.

In questo modo il programma può accedere ai suoi dati mediante indirizzamento indicizzato (RAM[FP+k], con k compreso tra 0 e 9), utilizzando celle di RAM il cui indirizzo sarà sempre compreso tra il valore contenuto in FP (0 nel nostro caso) ed il valore contenuto in SP (10 nel nostro caso), quest'ultimo escluso. L'estensione procedurale potrebbe avvenire, dal punto di vista della sovrapposizione dei dati nello stack, sostituendo al valore precedente di FP il valore precedente di SP, e poi sommando 15 al contenuto di SP (ottenendo quindi FP=10 ed SP=25 durante l'esecuzione di P2). Anche in questo caso, 15 è un valore "noto" a P2, mentre il valore di FP per P2 viene desunto dal valore di SP di P1.

Se P2 dovesse essere interrotto a sua volta per consentire l'esecuzione della estensione procedurale P3, la quale supponiamo richieda a sua volta 7 celle di memoria per la memorizzazione dei propri dati, allora potremmo sovrapporre una nuova fetta di memoria nello stack assegnando ad FP il valore precedente di SP (ossia FP=25) e poi incrementando il valore di SP di 7 (ottenendo quindi SP=32 durante l'esecuzione di P3). Il programma P3 potrebbe quindi anche lui accedere ai suoi dati locali mediante l'indirizzamento indicizzato RAM[FP+0], ..., RAM[FP+6], senza interferire coi dati di P1 e P2 e senza essere condizionato dal fatto che P1 e P2 sono stati interrotti temporaneamente per consentire alla CPU di eseguire P3.

Questo semplice meccanismo di sovrapposizione della memoria nello stack richiede inoltre di consentire l'operazione inversa di "rimozione" di uno strato dallo stack per ripristinare l'accesso corretto alle variabili di

un programma precedentemente interrotto. Nel nostro esempio, quando P2 termina la propria esecuzione, P1 deve poter continuare la propria esecuzione, recuperando la propria "fetta" di memoria. Poiché la porzione di memoria assegnata a un programma è sempre delimitata dai valori in FP e SP, è necessario che al termine di P2 tali valori siano ripristinati a quelli che delimitano l'area di memoria di P1. Mentre il "vecchio" valore da riassegnare al registro SP si trova ancora memorizzato nel registro FP (ossia, l'inizio della porzione di memoria di P2 coincide con la fine di quella di P1), il "vecchio" valore del registro FP (ossia, l'inizio della memoria di P1) sarebbe andato perso a seguito della sovrapposizione di una nuova fetta di memoria, e quindi non sarebbe più disponibile seguendo lo schema semplicistico delineato fin qua. La soluzione a questo problema può essere ottenuta in modo molto semplice copiando in una cella RAM il vecchio valore di FP prima che questo venga sostituito dal vecchio valore di SP durante l'attivazione della estensione procedurale. Ovviamente il posto più indicato per mantenere questo valore è una cella di RAM all'interno della struttura stack stessa, che viene posta in una zona "di confine" tra quella destinata al programma chiamante e quella destinata alla sua estensione procedurale. La procedura di sovrapposizione di una nuova fetta di memoria nello stack destinata a contenere i dati di un programma, attivato come estensione procedurale di un altro precedentemente in esecuzione, potrebbe quindi essere descritta dalla seguente sequenza di passi elementari:

```
RAM[SP] <- FP     // salva il vecchio valore di FP
FP <- SP          // aggiorna il valore di FP
SP <- (SP +k +1)  // aggiorna il valore di SP
```

dove "k" indica il numero di celle di RAM necessarie per contenere i dati della procedura che si sta attivando.

Analogamente, l'operazione di eliminazione dallo stack della fetta di memoria corrispondente ad una estensione procedurale conclusa (e quindi il ripristino dell'esecuzione del programma precedentemente interrotto) potrebbe essere ottenuta con la sequenza di passi elementari:

```
SP <- FP      // recupera il vecchio valore di SP
FP <- RAM[SP] // recupera il vecchio valore di FP
```

Dovrebbe risultare evidente, a questo punto, che la tecnica di strutturazione a stack della memoria per i dati dei programmi appena delineata risolve in parte il problema di voler sospendere l'esecuzione di un programma temporaneamente, con l'idea di voler poi riprendere successivamente l'esecuzione dello stesso programma dal punto in cui lo avevamo lasciato. In particolare, questa struttura risolve direttamente il problema dei valori calcolati e memorizzati in celle di memoria, mentre non risolve direttamente il problema dei dati parziali calcolati e memorizzati nei registri della CPU. Per questi ultimi, se il programma che sostituisce quello interrotto usa gli stessi registri per calcolare altri dati, occorre provvedere ad un "salvataggio" in memoria RAM prima dell'attivazione della estensione procedurale, e ad un successivo "ripristino" dei valori precedenti nei registri dopo la terminazione dell'estensione procedurale e prima del ripristino dell'esecuzione del programma interrotto. Tale salvataggio può ovviamente essere realizzato in ulteriori celle di memoria "di confine" appositamente predisposte nello stack (così come abbiamo già visto nel caso del registro FP).

Tra i valori dei registri da salvare per poter poi riprendere l'esecuzione del programma interrotto, dallo stesso punto in cui era stato interrotto, sicuramente dobbiamo considerare il registro contatore di programma

(PC). Tenendo conto di questa ulteriore necessità, le sequenze di passi elementari atte a realizzare le operazioni di attivazione e terminazione di una estensione procedurale (altrimenti dette *chiamata di procedura* e *ritorno da procedura*) potrebbero essere definite come segue.

Chiamata di procedura la cui prima istruzione si trova all'indirizzo `i` e che necessita di `k` celle nello stack per contenere i suoi dati:

```
RAM[SP] <- FP
RAM[SP+1] <- PC
FP <- SP
SP <- (SP +k +2)
PC <- i
```

Ritorno da procedura precedentemente attivata come sopra:

```
SP <- FP
FP <- RAM[SP]
PC <- RAM[SP+1]
```

Notiamo che quest'ultima modifica al meccanismo fa sì che il valore di PC salvato per il ritorno si venga a trovare nell'area di memoria della procedura chiamata che in questo modo "sa" come riattivare il programma chiamante al suo termine; ve notato anche che, per lo stesso motivo l'area di memoria dedicata ai dati propri della procedura parte in realtà da ll'indirizzo FP+1.

Discutiamo ora un'ultima estensione normalmente adottata dalle macchine convenzionali per rendere più conveniente la strutturazione a stack della parte di memoria dedicata ai dati. Spesso risulta conveniente strutturare un programma in modo da non dover allocare tutte le celle di memoria destinate a contenere dati fin dall'inizio dell'esecuzione. Sovente capita quindi che un programma in fase di esecuzione richieda l'uso di memoria non precedentemente allocata nella sua "fetta" di stack. L'aggiunta dinamica di nuove celle di memoria alla fetta di stack dedicata ad un programma P1 già in fase di esecuzione non comporta nessuna complicazione particolare, grazie all'uso dei due registri di delimitazione: basta incrementare il valore contenuto nel registro SP (lasciando inalterato il valore contenuto nel registro FP) per "inspessire la fetta" di stack destinata a P1; analogamente, basta decrementare il valore contenuto nel registro SP per 'assottigliare la fetta" di stack destinata a P1. Normalmente tutte le macchine convenzionali prevedono una istruzione chiamata PUSH che, usando il modo di indirizzamento auto-incremento, alloca una nuova cella nello stack e vi memorizza un valore. Analogamente una istruzione normalmente chiamata POP, sempre usando il modo di indirizzamento auto-incremento, legge il contenuto dell'ultima cella dello stack e la elimina (decrementando il valore contenuto nel registro SP). Vedremo in seguito come lo stack, con la sua struttura stratificata e le sue operazioni di *push* e *pop* costituisca una delle strutture dati fondamentale della programmazione, che viene utilizzata in moltissimi altri contesti oltre a quello visto nella strutturazione dell'uso della memoria.

A questo punto possiamo finalmente arrivare a comprendere come può essere realizzato il meccanismo di "chiamata di funzione con passaggio di parametri" mediante lo stack. L'uso del modo di indirizzamento

indicizzato non è vincolato a costanti di valore positivo: può tranquillamente funzionare anche sommando valori costanti negativi al contenuto del registro FP, consentendo quindi alla procedura chiamata di accedere allo fetta di stack nella quale sono memorizzati i dati del programma chiamante; l'unico vincolo, in questo senso, è legato alla "conoscenza del significato del contenuto dello stack" del programma chiamante da parte della procedura.

La condivisione delle stesse celle di memoria nello stack tra programma chiamante e procedura può essere realizzata adottando delle "convenzioni" note e accettate sia dai programmi chiamanti che dalle procedure. Per esempio, volendo programmare una funzione che calcola il prodotto tra due numeri interi e ne ritorna il risultato al programma chiamante (come visto nel caso della MVN), occorre adottare la seguente convenzione:

- Il programma chiamante (usando istruzioni PUSH) inserisce "in cima allo stack" tre nuove variabili corrispondenti ai valori dei due moltiplicandi (prima) e del risultato del prodotto (per ultimo);
- Il programma chiamante chiama la funzione di calcolo del prodotto;
- La funzione, una volta attivata, "sa di poter trovare" i due moltiplicandi nelle celle RAM[FP-2] e RAM[FP-3], e di dover memorizzare il risultato calcolato nella cella RAM[FP-1];
- Al termine dell'esecuzione della funzione, il programma principale riparte "sapendo" di poter recuperare il risultato del calcolo del prodotto eseguendo l'istruzione POP dallo stack.
- Se i due moltiplicandi non devono essere modificati dalla funzione, si può realizzare un meccanismo di passaggio dei parametri "per valore" che si realizza disallocando successivamente anche le due celle di RAM corrispondenti, ignorandone il contenuto.

Nel caso in cui si volesse invece consentire alla procedura chiamata di cambiare i valori dei parametri passati dal programma chiamante, si potrebbe modificare la convenzione stabilendo un passaggio dei parametri "per referenza". Nel caso di passaggio "per referenza" (o per riferimento), nello stack viene allocata una cella che contiene non il valore del parametro, ma l'indirizzo della cella di memoria nel quale tale valore è memorizzato. L'uso di un parametro "per referenza" da parte della procedura implica un doppio indirizzamento della RAM: in un primo tempo, accedendo alla cella RAM[FP-j] si trova l'indirizzo "i" dell'operando, e successivamente si accede alla cella RAM[i] (con modo di indirizzamento diretto).

Parte 5

Basi di programmazione imperativa

Per i contenuti di questo capitolo facciamo riferimento al libro

D.S. Malik, Programmazione in C++, 2011, Apogeo.

Il corso si propone di insegnare i principi della programmazione imperativa in generale. Il C++, ristretto alla sola parte inerente la programmazione imperativa, viene utilizzato come linguaggio di riferimento e per le esercitazioni, ma la maggior parte dei concetti di programmazione sono generali e adattabili agli altri linguaggi imperativi.

Un libro utile per approfondimenti è:

B. Stroustrup, Programming - Principles and practice using C++, 2009, Pearson education (disponibile solo in inglese)

Nel seguito viene fornito solo l'elenco degli argomenti oggetto di questo capitolo ed i riferimenti sui libri.

5.1 Primi passi

- Struttura di un programma (Malik, Capitolo 2, pp 27-31; Stroustrup Chapter 2)
- Tipi, operatori, espressioni (Malik, Capitolo 2, pp 32-45; Stroustrup, 3.8, 4.3)
- Variabili, costanti e assegnazione (Malik, Capitolo 2, pp.47-56; Stroustrup Chapter 3)

5.2 Input/Output

Per un uso di base di input da tastiera e output su schermo sono sufficienti poche nozioni:

- Output (Malik, Capitolo 2, pp. 56-61)
- Input (Malik, Capitolo 3, pp. 82-85)

Si consiglia comunque la lettura di tutto il Capitolo 3 del Malik per dettagli che sono rilevanti nell'uso pratico. Il Malik è piuttosto carente sull'uso dei file: il minimo indispensabile si trova alle pp.114-118, mentre i capitoli 10 e 11 dello Stroustrup trattano l'argomento in modo esauriente.

5.3 Strutture di controllo

- Selezione: `if`, `switch` (Malik, Capitolo 4; Stroustrup, 4.4.1)
- Cicli: `while`, `for` (Malik, Capitolo 5; Stroustrup 4.4.2)

5.4 Funzioni

Malik, Capitolo 6; Stroustrup 4.5.

È molto importante approfondire la parte relativa al passaggio dei parametri e all'allocazione della memoria sullo stack, anche in relazione ai contenuti del Capitolo 3 di queste dispense.

5.5 Array statici

Malik, Capitolo 7

5.6 Il costruttore di tipi `struct`

I linguaggi di programmazione di alto livello permettono al programmatore di definire i propri tipi. Lo schema array fornisce un costruttore per aggregare dati omogenei (N.B.: in linguaggi come il Pascal, l'array è un vero e proprio costruttore di tipi; tuttavia, in C e C++ non è possibile definire un "tipo array" ma solo variabili array; al contrario la classe tamplate vector del C++ che vedremo nel prossimo capitolo è un vero costruttore di tipi, parametrico sul tipo base della singola cella).

L'aggregazione di dati disomogenei tra loro richiede un costruttore di tipi diverso. In C e C++ questo costruttore si chiama `struct` e consente di definire la struttura di contenitori atti a contenere dati di diverso tipo. Possiamo pensare ad un tale contenitore come una "scatola" con diversi "scomparti", ciascuno di forma e dimensioni tali da contenere un certo tipo di oggetti e non altri: sarà possibile trattare tale scatola come un

tutt'uno, ad esempio per spostarla da un posto all'altro, oppure andare in un singolo scomparto per vedere cosa contiene (leggere) o metterci qualcosa dentro (scrivere); a questo scopo è opportuno dare a ciascun scomparto un nome, in modo da potervi accedere agevolmente. Notiamo qui come il meccanismo differisca da quello degli array: in quel caso non è necessario dare un nome agli scomparti perché, essendo questi tutti uguali, è sufficiente numerarli e accedere ad un dato scomparto attraverso il suo indice.

I vari "scomparti" di una `struct` si chiamano *campi.* A livello linguistico la definizione di un tipo `struct` ha lo schema seguente:

```
struct nomestruct {
      tipo1 campo1;
      tipo2 campo2;
      ........
      tipon campon;
};
```

In tutto ciò, l'unica parola chiave è `struct`, mentre gli altri sono identificatori definiti dall'utente: `tipo1`, ..., `tipon` sono nomi di tipi, che devono essere noti all'atto della definizione della struct, `nomestruct` è il nome dato al tipo struct stesso e `campo1`, ..., `campon` sono i nomi dei singoli campi che contiene.

Parte 6

Strutture dinamiche

I dati trattati fino a questo momento sono sempre mantenuti in variabili globali o locali, secondo le definizioni date in Sezione 3.5, che sono allocate rispettivamente nell'area delle variabili globali e sullo stack. In molti casi, tuttavia, può essere necessario manipolare strutture dinamiche. In linea generale, queste sono necessarie tutte le volte che la quantità di dati da mantenere in memoria non è nota durante la progettazione del programma. Ad esempio, non è possibile scrivere un programma che immagazzina un insieme di dati omogenei in un array statico senza sapere a priori quale sarà il numero massimo di questi dati. Si potrebbe scegliere a priori una dimensione grande per l'array, in modo da poter gestire input di elevata numerosità. Tuttavia, questa scelta sprecherà molta memoria (preziosa ad esempio per l'esecuzione degli altri processi) nel caso in cui l'input contenesse pochi dati, mentre potrebbe risultare insufficiente nel caso di un input molto grande. La scelta più efficace, quando non si possono fare assunzioni a priori sulla quantità dei dati in input, è di consentire al programma di allocare *dinamicamente*, cioè mentre gira, lo spazio necessario per i propri scopi. Come abbiamo già anticipato in Sezione 3.5, le variabili dinamiche saranno allocate sullo heap e il programma si dovrà preoccupare di gestire esplicitamente l'allocazione della memoria dinamica e la sua deallocazione una volta che questa diventi inutile.

Nelle sezioni seguenti, presenteremo il meccanismo generale di gestione della memoria dinamica in C++ e discuteremo due delle principali classi di strutture dati dinamiche: gli array dinamici e le liste. Nel corso di Algoritmi e Strutture Dati tali argomenti saranno estesi ulteriormente: ad esempio, con gli stessi strumenti di base utilizzati per confezionare le liste si vedrà come è possibile trattare strutture dati molto più complesse, come gli alberi e i grafi.

6.1 Puntatori

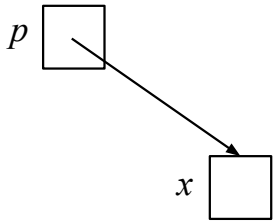
Un *puntatore* è una variabile atta a contenere come valore l'indirizzo di un'altra variabile ed è sempre associato ad un tipo: un puntatore al tipo `T` può contenere l'indirizzo di qualunque variabile di tipo `T` e di nessun altro tipo. La dichiarazione di una variabile puntatore quindi richiede di specificare tre cose: il *nome* della variabile,

il fatto che questa sia un *puntatore* e il *tipo* a cui si riferisce. In C e C++ questo si realizza inserendo il simbolo `*` tra tipo e nome nella dichiarazione:

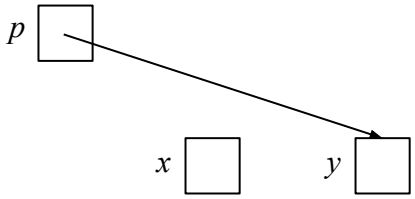
```
int * p;
```

indica che `p` è una variabile puntatore atta a mantenere indirizzi di variabili di tipo `int`. Un altro modo di vedere le cose è che `p` sia di tipo `int*`, ossia interpretare l'asterisco come un "modificatore" che crea un tipo puntatore a partire da un tipo dato. Vedremo che la possibilità di associare l'asterisco con il tipo alla sua sinistra (per definire un tipo puntatore) o con la variabile alla sua destra (per riferire una variabile puntatore a un dato tipo) è un buon modo per memorizzare le cose - ma può anche essere fonte di grande confusione se non si sono capite bene.

Se `x` è una variabile di tipo `T` , `p` è un puntatore a `T` (ossia `p` è di tipo `T*`) e il valore di `p` coincide con l'indirizzo di `x`, si dice che `p` *punta* a `x` e si usa il formalismo grafico seguente.



Una variabile puntatore `p` può puntare in un dato momento ad una certa variabile (quindi a una certa area di memoria) ed in un altro momento ad un'altra. Nel formalismo grafico, cambiare il valore di `p` in modo che punti a un'altra variabile `y` corrisponderà a "spostare la freccia" in modo che la sua punta vada a toccare la casella corrispondente a `y`; in realtà questo significa cambiare il contenuto di `p`, scrivendoci dentro l'indirizzo di `y`.



A livello di linguaggio, per consentire a questo meccanismo di funzionare sono necessari due operatori:

- *Operatore di referenziazione:* data una variabile `x`, restituisce un valore puntatore, corrispondente all'indirizzo di `x`. In C e C++ questo operatore si realizza con il simbolo `&` pertanto l'espressione `&x` restituisce un valore puntatore corrispondente all'indirizzo di `x` (per brevità diremo che restituisce l'indirizzo di `x`, identificando da ora in poi indirizzi e puntatori). L'operatore `&` si può usare ad esempio nelle assegnazioni per assegnare ad un puntatore l'indirizzo di una variabile. L'istruzione

  ```
  p = &x;
  ```

assegna a p l'indirizzo di x.

- *Operatore di dereferenziazione:* dato un puntatore p, restituisce la variabile puntata da p. In C e in C++ questo operatore si realizza con l'operatore * (da non confondersi con l'operatore di moltiplicazione), pertanto l'espressione *p restituisce la variabile puntata da p. Tale variabile sarà valida solo se il valore di p punta ad un indirizzo di memoria valido, cioè ad un indirizzo già allocato dal programma. Sono validi gli indirizzi delle variabili globali o statiche, delle variabili locali visibili e delle variabili dinamiche precedentemente allocate; queste ultime sono spiegate nella prossima sezione.

  Notiamo la consistenza dell'operatore di dereferenziazione con la dichiarazione di puntatore: la dichiarazione `T * p;` significa al tempo stesso che p è di tipo puntatore T* e che *p è una variabile di tipo T (purché il valore di p sia valido).

È importantissimo capire che, mentre l'operatore di referenziazione & produce sempre un valore destro (l'indirizzo della variabile), il puntatore dereferenziato *p può essere usato sia come valore destro (il contenuto della variabile puntata da p) che come valore sinistro (la variabile stessa). Quindi, se p punta ad un indirizzo di memoria valido, sarà possibile variare il contenuto della variabile puntata da p, ad esempio con un'assegnazione

```
*p = <exp>;
```

dove <exp> denota una qualunque espressione il cui risultato sia compatibile con il tipo puntato da p. Quindi:

> *un'assegnazione fatta su* p *fa sì che* p *vada a puntare da un'altra parte ("sposta la freccia") mentre un'assegnazione fatta su* *p *cambia il contenuto della cella puntata da* p.

Si veda Figura 6.1 per un esempio. Questo vale non solo per l'assegnazione ma per tutti i contesti in cui *p assuma ruolo di valore sinistro.

Attenzione: Se p punta a un indirizzo di memoria non valido il programma normalmente abortisce, sia che si tenti di usarlo come valore destro, che come valore sinistro. Tuttavia non sempre questo accade: un puntatore non inizializzato o aggiornato in modo non corretto potrebbe puntare comunque ad un'area allocata. Tentare di leggere da quell'area di solito produrrà risultati privi di senso, mentre tentare di scriverci dentro ne corromperà il contenuto. Spesso in questi casi il programma prosegue nel suo percorso ma produce risultati errati e molte volte abortisce in un momento successivo a causa dell'uso di dati corrotti precedentemente. Trovare la causa di questo tipo di errori può essere molto difficile, pertanto bisogna fare la massima attenzione quando si manipolano i puntatori.

6.2 Allocazione e deallocazione dinamica

L'uso dei puntatori per far riferimento a variabili allocate in modo statico è poco interessante e può essere evitato nella maggior parte dei casi. I puntatori sono invece di primaria importanza nella gestione della memoria

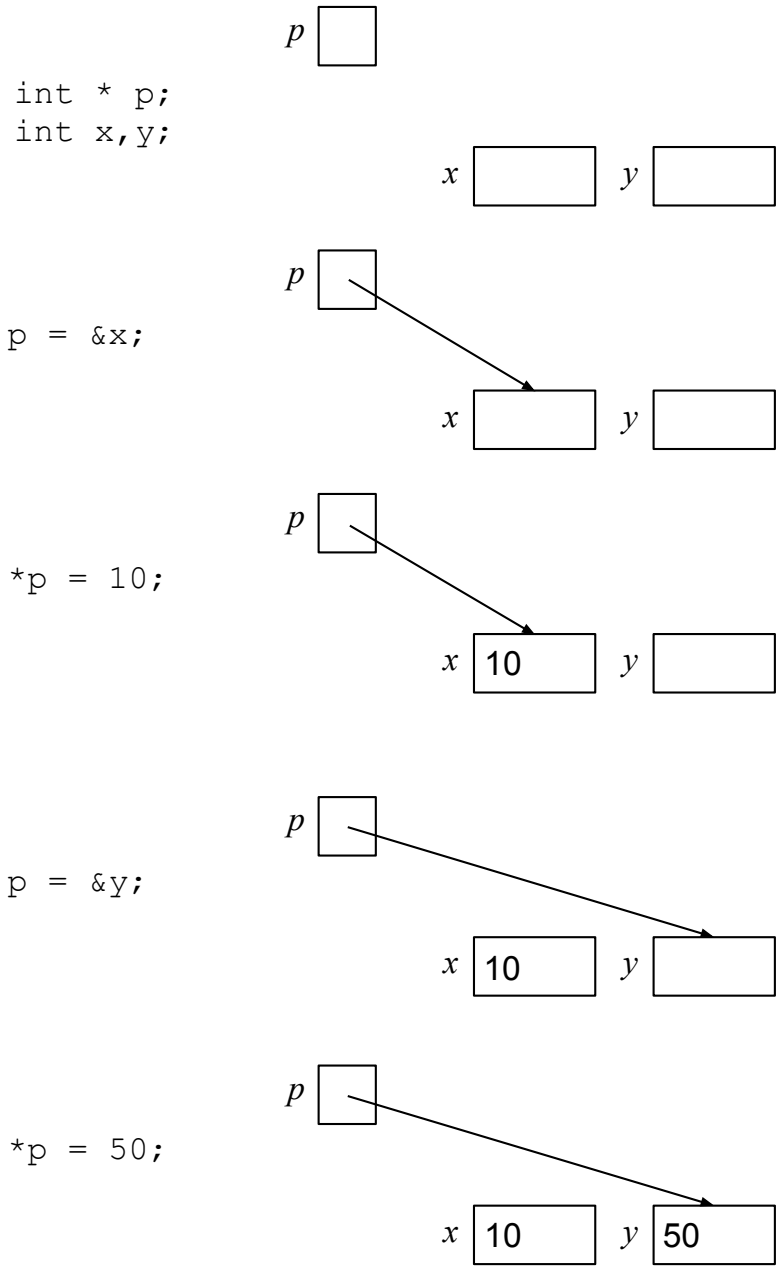


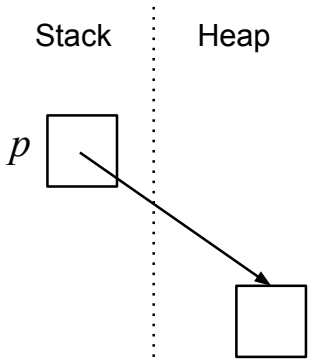
Figura 6.1: Un frammento di codice C/C++ (sulla sinistra) e la relativa evoluzione della memoria (sulla destra). Le caselle vuote indicano valori indefiniti.

dinamica. Questa necessita di due operazioni fondamentali, una per allocare memoria sullo heap e l'altra per restituire memoria allo heap:

- *Allocazione dinamica:* chiede al gestore dello heap di allocare la memoria necessaria per mantenere un dato di un certo tipo; il gestore alloca la memoria necessaria sullo heap e ne restituisce l'indirizzo. In C++ questa operazione si realizza attraverso il comando `new`. Se `p` è un puntatore di tipo `T`, l'istruzione

```
p = new T;
```

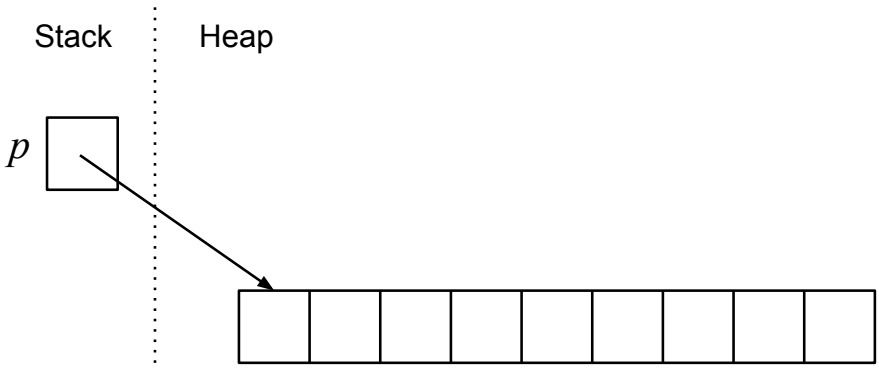
alloca sullo heap lo spazio necessario per mantenere una variabile di tipo `T` e scrive in `p` l'indirizzo di tale variabile. Da quel momento - e finché il valore di `p` non venga cambiato - la variabile `*p` rappresenta in tutto e per tutto l'area di memoria appena allocata ed il suo contenuto. Notiamo che mentre `p` vive sullo stack (in quanto variabile con un nome), la memoria a cui questa punta vive sullo heap. Possiamo quindi associare la seguente notazione grafica all'assegnazione di cui sopra, in cui distinguiamo tra stack e heap.



È anche possibile allocare dinamicamente array con un comando del tipo

```
p = new T[size];
```

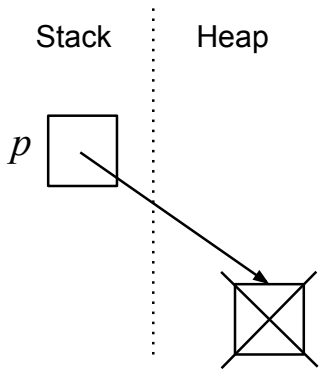
dove `size` può essere una costante, una variabile o un'espressione intera, che denota il numero di celle dell'array. Questa operazione alloca sullo heap lo spazio necessario per mantenere in locazioni consecutive `size` valori di tipo `T`; il puntatore `p` punterà alla prima locazione di tale array. Vedremo di seguito che tutte le celle dell'array sono indirizzabili attraverso `p`, come nel caso degli array statici. La figura seguente illustra questa situazione mediante il formalismo grafico.



- *Deallocazione dinamica:* restituisce allo heap una porzione di memoria dinamica che non serve più al programma; tale operazione invalida l'accesso a quella porzione di nemoria e la lascia a disposizione del gestore dello heap per soddisfare richieste successive. In C++ questa operazione si realizza attraverso il comando `delete` che esiste in due varianti. Se `p` è un puntatore di tipo `T`, l'istruzione

```
delete p;
```

restituisce allo heap la locazione di memoria puntata da `p`. Notiamo che viene specificato un puntatore, ma l'operazione agisce sulla cella puntata, non sul puntatore stesso. Il formalismo grafico viene illustrato di seguito.



Dopo aver effettuato la `delete`, la cella barrata non è più valida/accessibile ed è stata restituita allo spazio libero nello heap, mentre il valore di `p` resta invariato e a quel punto `p` punterà ad una locazione non valida. Quindi sarà necessario assegnare un nuovo valore a `p` prima di utilizzarlo come valore destro.

Nel caso `p` punti ad un array dinamico (quindi un blocco di diverse celle consecutive del tipo puntato da `p`), per deallocare l'intero array è necessario utilizzare la seconda variante dell'operazione di dellocazione:

```
delete[] p;
```

Notiamo che in questo caso non è necessario specificare il numero di celle da deallocare: il runtime environment "ricorda" quante celle consecutive erano state allocate insieme a partire da quella puntata da `p` e le dealloca tutte.

Attenzione: mentre l'allocazione dinamica è imprescindibile per utilizzare lo heap, a prima vista la deallocazione della memoria dinamica può sembrare un'operazione superflua, tanto più che questa memoria viene comunque liberata una volta che il programma termina la propria esecuzione. In realtà, si tratta di un'operazione importante e delicata, che va gestita con molta attenzione, anhe se la sua utilità sarà meglio appressata in corsi più avanzati. La mancata deallocazione di aree di memoria non più raggiungibile viene detta *memory leak*: crea zone di heap inutilizzate e non riutilizzabili e può portare un programma ad esaurire rapidamente la memoria a disposizione, con esiti catastrofici.

6.3 Array dinamici

Gli array dinamici si possono usare con la stessa sintassi di quelli statici e necessitano delle stesse precauzioni. Dato un puntatore `p` a cui sia stato assegnato un blocco di memoria corrispondente ad un array (cioè un gruppo

di celle del tipo puntato da p), nel codice successivo sarà possibile accedere alla cella di indice i dell'array con il simbolo p[i] e tale simbolo si potrà usare sia come valore destro che come valore sinistro. Consideriamo il seguente blocco di codice:

```
int a[10];
int * p;
p = new int[10];
```

Abbiamo qui un array statico a ed un array dinamico p della stessa dimensione. Visti come valori destri, sia a che p sono puntatori di tipo int*, che puntano entrambi alla prima cella del rispettivo array di 10 interi. Le celle di ciascuno di questi array possono essere usate sia come valori destri che come valori sinistri attraverso il meccanismo solito con le parentesi quadre. Le differenze sostanziali tra i due sono:

- l'array puntato da p risiede nello heap, mentre quello puntato da a risiede sullo stack;
- nella dichiarazione dell'array puntato da a è obbligatorio specificare una dimensione costante (infatti la memoria necessaria sullo stack viene decisa a compile time), mentre nell'allocazione dinamica dell'array puntato da p si potrebbe usare anche un'espressione variabile, il cui valore può variare volta per volta in funzione dell'input del programma;
- l'associazione (*binding*) tra a e il rispettivo array è statica (infatti a è un "puntatore costante"), mentre quella tra p e il rispettivo array è dinamica: è stata decisa dal programma durante la sua esecuzione e p potrebbe puntare a qualcos'altro in un altro punto del programma.

Come nel caso statico, anche nel caso dinamico l'indirizzamento di una cella fuori dai limiti dell'array può avere effetti impredicibili: se a fronte di un'allocazione p = new int[10] poi cerchiamo di accedere a p[12] il programma tenta di accedere ad una cella di tipo int che si trova esattamente 12 "posti" dopo quella puntata da p. Possono succedere due cose, in alternativa:

- Se tale cella si trova fuori dall'area allocata dallo heap, il programma abortisce;
- Se tale cella si trova dentro l'area allocata, il programma prosegue, ma l'eventuale valore ottenuto in lettura sarà privo di senso, mentre l'eventuale valore scritto avrà corrotto un'area di memora esterna all'array, con probabili conseguenze negative e spesso fatali sulla prosecuzione del programma.

Mediante gli array dinamici è possibile implementare contenitori "flessibili", la cui capacità si adatti dinamicamente alle esigenze del programma. Immaginiamo ad esempio di dover mantenere un numero indeterminato di valori omogenei di un tipo noto T. Tali valori vengono acquisiti dal programma come input durante il suo funzionamento e non arrivano tutti insieme, ma arrivano alla spicciolata e devono essere messi tutti nello stesso contenitore. Il programmatore non ha conoscenza di quale sarà il numero di valori letti dal programma, quindi non può pensare di utilizzare un array statico. Neppure il programma stesso mentre gira sa prevedere quanti dati arriveranno nell'istante in cui arriva il primo di questi. La soluzione sta nel meccanismo seguente, che è molto utile implementare come esercizio:

1. si alloca inizialmente un array dinamico `p` di piccola dimensione (ad esempio 8 celle) e si ricorda la sua dimensione in una variabile intera `s`;
2. si mantiene un'altra variabile intera `n` che conta quante sono le celle di `p` occupate, inizialmente zero;
3. ogni volta che arriva un nuovo dato, si tenta di inserirlo nell'ultima locazione libera (`p[n]`) e si incrementa `n`;
4. tuttavia, se `n` e `s` hanno lo stesso valore, questo non è possibile (non c'è più spazio); in quel caso:
   (a) si estende l'array allocandone uno di dimensione maggiore (ad esempio doppia);
   (b) si ricopia l'array puntato da `p` nella prima parte del nuovo array;
   (c) si mette il nuovo dato nella cella successiva all'ultima occupata;
   (d) si aggiornano i valori di `n` e `s`;
   (e) si cancella (mediante `delete[]`) l'array puntato da `p` e si fa puntare `p` al nuovo array.

Si può seguire una politica analoga per permettere di cancellare elementi in coda all'array: per cancellare un elemento si decrementa semplicemente il contatore `s`; quando questo scende sotto la metà della dimensione dell'array, si dimezza lo spazio allocato.

Aritmetica dei puntatori. I puntatori possono essere usati anche come cursori, attraverso operazioni che consentono di posizionarli su celle diverse dalla prima. Ogni puntatore può essere trattato come un numero intero, sul quale si possono effettuare incrementi, decrementi, somme e sottrazioni: ogni unità di incremento/decremento di un puntatore corrisponde a spostarlo di una cella avanti/indietro nell'array a cui punta. L'aritmetica dei puntatori può risultare molto comoda in certi contesti, ma va detto che presenta alcuni lati delicati e che si può comunque fare tutto senza utilizzarla, ad esempio mediante indici interi come si fa negli array statici. Per questo motivo, non approfondiremo oltre questo aspetto.

6.4 I vector del C++

La libreria standard del C++ mette a disposizione diversi *contenitori*, tra cui il `vector` che implementa il meccanismo descritto nella sezione precedente in un contesto molto controllato e molto facile da usare. I vector sono un esempio di *classe template*, un tipo di costrutto molto avanzato del C++ che non sarà sviluppato ulteriormente nel contesto di questo corso. Pertanto, ci accontenteremo di utilizzarli senza farci troppe domande su come sono costruiti. Basti sapere che la loro implementazione è effettivamente basata su array dinamici e meccanismi simili a quelli appena descritti, benché molto più sofisticati. Tali meccanismi permettono ai vector di poter essere utilizzati con ragionevole tranquillità, rispetto alla definizione di array dinamico che abbiamo precedentemente elencato. In particolare:

- ci permettono di contenere con maggiore facilità gli accessi fuori dalla memoria (out of bound)

- forniscono meccanismi semplici per realizzare copie profonde tra vector.

L'uso di vector prevede l'inclusione del corrispondente modulo di libreria con la direttiva `#include <vector>`. La dichiarazione di una variabile vector prevede la specifica del tipo base (quello di ogni cella del vector) con una sintassi particolare:

```
vector<T> v;
```

indica la dichiarazione di una variabile `v` che è un vector di celle di tipo `T`, dove `T` deve essere un tipo noto. In questa dichiarazione il vettore è inizialmente vuoto, nel senso che è composto di zero celle. È anche possibile specificare un vector di dimensione data mediante la dichiarazione `vector<T> v(n);` in cui `n` è una variabile, costante o espressione intera che specifica le celle disponibili in `v`; in quest'ultimo caso `v` è composto di `n` celle non inizializzate (le celle esistono ma i loro valori sono indefiniti). Le celle in un vettore `v` si possono indirizzare mediante la notazione a parentesi quadre come quelle degli array. Inoltre sono disponibili diverse funzioni che permettono una manipolazione facile dei vector. Queste funzioni utilizzano la *dot notation*, secondo cui la funzione che opera su un dato si specifica attraverso il nome del dato, seguito da un punto e quindi dal nome della funzione stessa. Elenchiamo di seguito le funzioni più comuni, rimandando al Malik (Capitolo 11 pp.552-571) e allo Stroustrup (Appendix B.4, pp.1107-1111) per ulteriori dettagli. Dato il vector `v`:

- `v.size()` restituisce il numero di celle presenti in `v` (quindi un vector, contrariamente ad un array, "sa" di quante celle è costituito);
- `v.at(i)` ha lo stesso significato di `v[i]` ma permette di accedere solo a memoria allocata; un tentativo di accesso a memoria fuori dai limiti del vector fatto attraverso la funzione `at` non ha effetto (nel senso che non produce un valore e non modifica il vector stesso) ma genera un codice di errore che può essere verificato attraverso altre funzioni apposite; questo impedisce che il programma abortisca a causa di accessi sbagliati alla memoria, tuttavia resta compito del programmatore verificare che tali tentativi di accesso non si verifichino e trattare le eventuali eccezioni qualora questo accada;
- `v.resize(n)` ridimensiona `v` ad una dimensione di `n` celle; `n` può essere sia maggiore che minore del valore attuale;
- `v.push_back(x)` aggiunge una cella contenente il valore `x` in coda a `v`, aumentando di una unità il numero di celle disponibili di `v`.

Attraverso l'operazione di *push back* è possibile riempire facilmente un vector inizialmente vuoto aggiungendo un elemento alla volta in coda, in modo da ottenere un vector che sia esattamente della dimensione desiderata; quando invece sia noto già dal principio il numero dei dati da memorizzare nel vector (ad esempio perché questo numero viene letto da file prima di leggere i dati stessi), può essere conveniente allocare direttamente il vector della dimensione giusta (tramite la dichiarazione con dimensione oppure usando la funzione *resize* una sola volta); la scansione di un vettore può essere regolata facilmente da un cursore che parte dalla cella 0 e prosegue fino al valore precedente a quello di *size*; ecc.

6.5 Liste

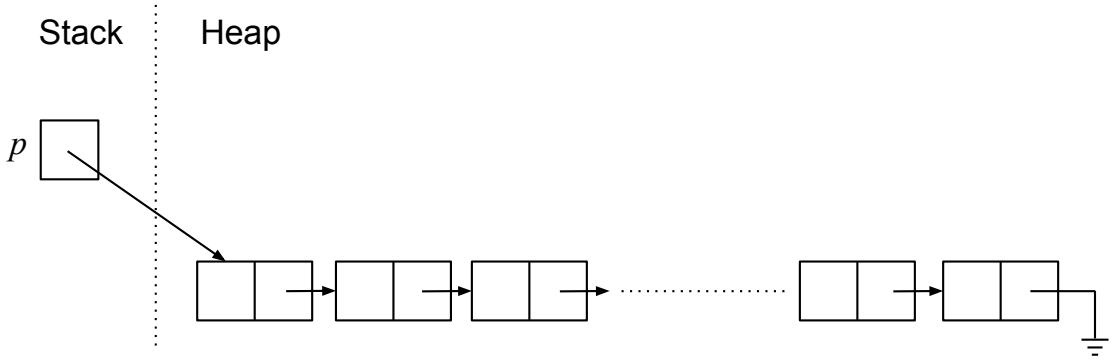
Benché gli array (statici o dinamici) forniscano la struttura dati principale per il trattamento di sequenze di dati omogenei, in alcuni casi possono risultare scomodi o inefficienti da usare:

- Alcune operazioni, come l'inserimento o la cancellazione di elementi in posizione intermedia, sono laboriose e computazionalmente costose: l'inserimento di un elemento richiede di "fare posto" spingendo avanti di una posizione tutti gli elementi successivi; analogamente, la cancellazione richiede di "recuperare posto" tirando indietro di una posizione tutti gli elementi successivi;

- Quando la quantità degli elementi da gestire varia notevolmente nel corso del programma, la gestione "elastica" degli array dinamici non solo è necessaria, ma può richiedere di fare copie dei dati tanto frequentemente da rendere inefficiente il codice. Va detto che questo problema in C++ è notevolmente mitigato dall'implementazione molto efficiente dei vector, quindi spesso si preferisce usare vector anche in presenza di sequenze "molto elastiche".

In questi casi, può essere conveniente utilizzare le *liste*, che forniscono un valido supporto alla gestione dinamica dei dati, anche se perdono alcune delle buone caratteristiche presenti negli array. In particolare *si perde l'accesso diretto a locazioni arbitrarie mediante indici*.

Le liste si realizzano mediante concatenazione, attraverso puntatori, di singole celle contenenti i dati. Ogni cella contiene un dato, viene allocata separatamente nella memoria dinamica e, nel caso base della *lista semplice*, contiene anche un puntatore che serve ad individuare la prossima cella nella sequenza. È immediato capire la struttura di una lista semplice dal formalismo grafico presentato di seguito.



Come per gli array dinamici, l'accesso alla testa della lista viene fornito da un puntatore. Ogni cella della lista è costituita di due parti: il contenitore per il dato (vuoto in figura) e quello per il puntatore alla prossima cella. I puntatori quindi forniscono gli "anelli" che tengono insieme la "catena", cioè la sequenza di dati. La sequenza può essere facilmente manipolata per inserire/cancellare elementi oppure cambiarne l'ordinamento, semplicemente cambiando il valore di tali puntatori. I puntini indicano che la stessa struttura viene ripetuta per tutte le celle presenti, che possono essere in numero arbitrario e non fissato a priori.

Il simbolo della "terra" (preso dalla simbologia dei circuiti elettrici) presente sull'ultima cella rappresenta il *puntatore nullo*: questo è un valore particolare per rappresentare un puntatore che "non punta da nessuna parte", da non confondersi con il puntatore indefinito, che potrebbe puntare da qualunque parte, incluse le locazioni non valide. Il puntatore nullo ha una funzione importantissima di "tappo", simile al carattere `'\0'` nelle stringhe, che nella lista serve a identificare il fatto che non c'è nessuna cella che segue l'ultima, ossia la lista finisce lì. In C++ il puntatore nullo si denota con la costante `nullptr` o con l'intero 0 e questi simboli denotano il puntatore nullo per qualunque tipo puntatore (cioè indipendentemente dal tipo base puntato).

Notiamo che ciascuna cella della lista "aggrega" due dati non omogenei: un valore del tipo base `T` e un puntatore al tipo corrispondente alla cella stessa.

6.5.1 Celle

Per definire il tipo delle celle nelle liste semplici di cui sopra, si può usare la definizione seguente di cella, il singolo elemento della lista:

```
struct cell {
      T info;
      cell * next;
};
```

dove `T` deve essere un tipo noto che rappresenta il tipo di contenuto di ciascuna cella, accessibile mediante il campo `info`. Notiamo che la definizione del campo `next`, che ha lo scopo di puntare alla prossima cella, necessiti del tipo `cell` stesso che è in via di definizione.

Nel codice che segue la definizione di struct (limitatamente allo scope in cui questa vive) il programma ha a disposizione un nuovo tipo corrispondente al nome della struct, quindi è possibile dichiarare variabili di quel tipo. Ad esempio, la dichiarazione `cell c;` alloca (sullo stack) una variabile `c` di tipo `cell`, i cui campi possono essere raggiunti mediante la *dot notation* `c.info` e `c.next`, rispettivamente. I campi denotati tramite questa notazione possono essere usati sia come valori destri che come valori sinistri.

Per utilizzare le struct in un contesto dinamico, allocandole sullo heap, è necessario agganciarle mediante puntatori. Se noi dichiariamo quindi `cell * p;` abbiamo allocato (sullo stack) un puntatore a celle ed è possibile ad esempio allocare una cella sullo heap attraverso l'istruzione `p = new cell;`. Per raggiungere i campi di una cella puntata da un puntatore `p` è possibile combinare l'operatore di dereferenziazione con la dot notation, scrivendo ad esempio `(*p).info` e `(*p).next`. Tuttavia è molto più comune utilizzare la *arrow notation* alternativa, che mette in evidenza il fatto che si sta accedendo mediante puntatori. Nel nostro caso, si scriverà `p->info` e `p->next` per raggiungere rispettivamente i due campi della struct puntata da `p`. La freccia `->` si legge *"puntato"*.

6.5.2 Liste semplici

Abbiamo ora tutti gli strumenti necessari per definire e manipolare le liste semplici, come definite in precedenza. Per la costruzione di queste liste definiamo i tipi seguenti:

```
struct cell {
      T info;
      cell * next;
};
typedef cell * list;
```

Si noti che il secondo tipo, definito mediante il costruttore `typedef`, altro non è che una ridenominazione del tipo puntatore a `cell`. Risulta però utile distinguere, concettualmente, tra il singolo puntatore, usato come "anello" della catena che costituisce la lista, e l'intera lista, che è comunque accessibile mediante un puntatore dello stesso tipo. La dichiarazione (con inizializzazione)

```
list l = nullptr;
```

istanzia una lista vuota. Da qui in poi, possiamo pensare alla variabile `l` come all'intera lista, anche se sappiamo che, a basso livello, altro non è che un singolo puntatore che permette l'accesso alla lista stessa. Inserire un elemento `x` di tipo `T` in testa alla lista `l` è un'operazione molto semplice (contrariamente a quanto accadrebbe con gli array che richiedono di spostare tutti gli elementi in avanti di un posto), che si può effettuare come illustrato in Figura 6.2. Notiamo che inserendo in testa in questo modo a partire da una lista vuota, il campo next dell'ultima cella punterà sempre a null.

Se una lista `l` contiene almeno un elemento, è possibile accedere al primo elemento della lista tramite `l->info`. Tutavia, l'operazione di dereferenziazione tramite la arrow notation non è valida se `l` vale nullptr : se si tenta di eseguire un accesso al campo di una cella a partire da un puntatore nullo il programma abortisce.

Tramite puntatori ausiliari è possibile scandire una lista in modo da leggere i suoi elementi in sequenza. Immaginiamo di avere una lista `l` piena di elementi e una funzione `leggi` che genericamente fa un'operazione "in lettura" su un valore di tipo `T`. Lo schema di scansione sequenziale della lista è dato dal frammento di codice seguente:

```
cell * aux = l;     // puntatore ausiliario all'inizio
while (aux!=nullptr)   // cicla fino alla fine
{
  leggi(aux->info); // legge dalla cella corrente
  aux = aux->next;  // salta alla cella successiva
}
```

Notiamo che durante la scansione la lista non viene modificata, mentre il puntatore `aux` svolge funzione di cursore (come gli indici interi negli array). Diversamente dagli array, in cui il cursore può essere spostato

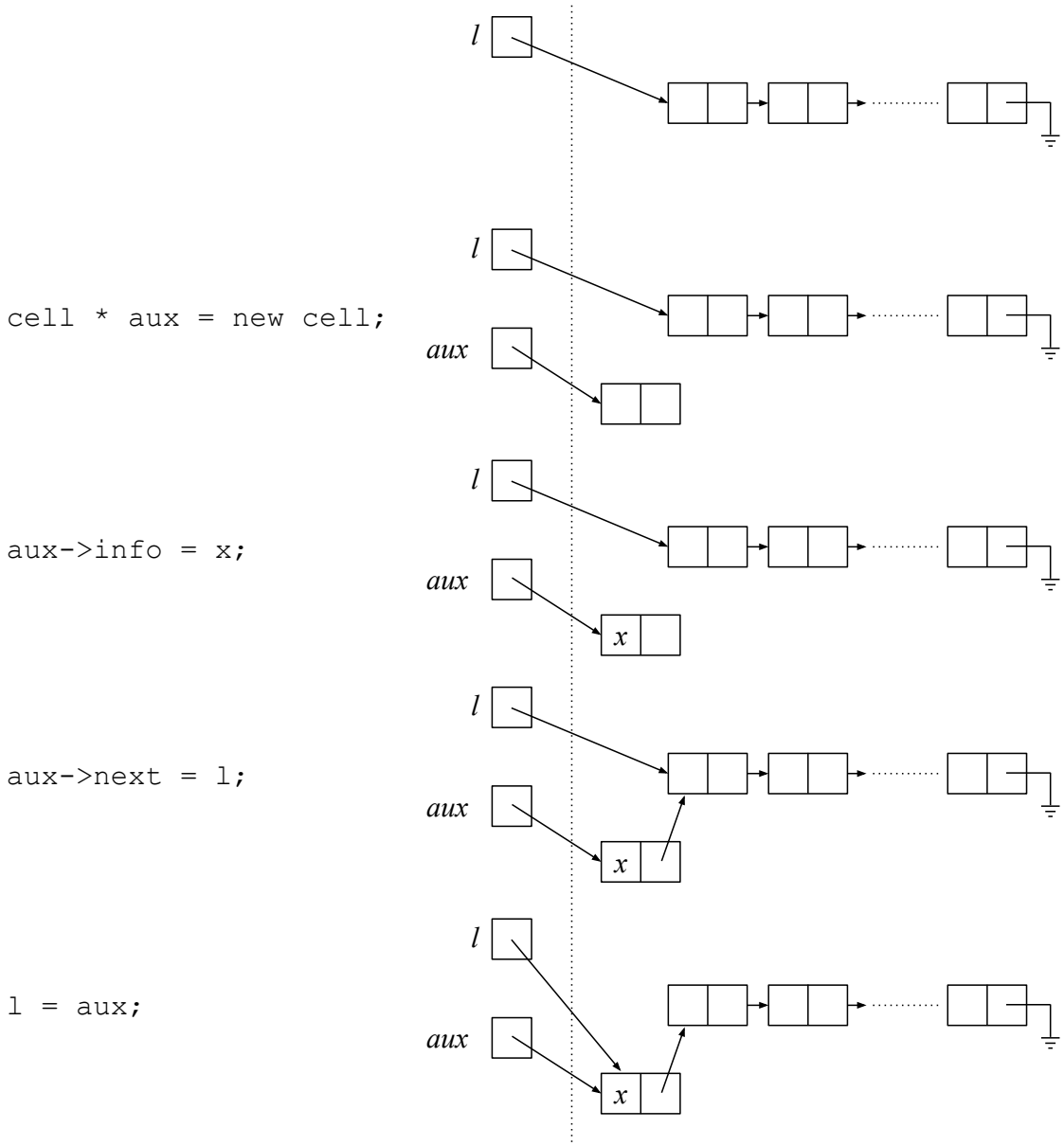


Figura 6.2: Inserimento di un elemento x in testa ad una lista l.

semplicemente incrementando il suo valore, nelle liste lo spostamento richiede il cambio esplicito di puntatore attraverso il campo next. Notiamo come, rispetto agli array, sia andata persa la possibilità di accedere dirittamente alla cella in una data posizione: se vogliamo saltare alla n-esima cella di una lista semplice, sarà necessario scandirla dall'inizio effettuando n passi.

La ricerca di un elemento si fa con lo stesso codice della scansione, solo che il ciclo si arresta, in alternativa, se l'elemento viene trovato oppure se si raggiunge il fondo della lista; se la lista è ordinata si può interrompere il ciclo anche se si incontra un elemento di valore maggiore di quello cercato (provare per esercizio).

L'inserimento di un elemento in un dato punto della lista è più laborioso e consiste di tre fasi:

1. Preparare la cella da inserire, come nel caso dell'inserimento in testa;
2. Scandire la lista per trovare il posto in cui inserire (in base ad un dato criterio, ad esempio: inserire al posto i-esimo, oppure inserire prima/dopo un dato valore, ...);
3. Riconfigurare i puntatori della nuova cella e di quella che la precede in modo da sganciare e riagganciare le celle per realizzare l'inserimento.

Nel realizzare le fasi 2 e 3 bisogna fare molta attenzione a trattare nel modo corretto i casi in cui l'inserimento vada fatto al primo o ultimo posto, incluso il caso in cui la lista sia vuota. Inoltre, per poter effettuare la fase 3 è necessario che durante la fase 2 non vada perso l'accesso (puntatore) alla cella che precede quella da inserire. La scansione si può effettuare con un ciclo simile a quello già visto. In questo caso però il ciclo si deve arrestare per due possibili condizioni: o si è trovato il punto di inserimento, oppure la lista è finita (e allora il punto di inserimento è il fondo della lista stessa). Realizzando la scansione con un cursore, come già visto, le cose si complicano quando l'inserimento vada fatto *prima* di una data cella (ad esempio la cella che contiene un dato valore, oppure la prima cella che contiene un valore maggiore di quello da inserire): quando la scansione arriva al punto cercato, il riferimento alla cella precedente è andato perso, perché le liste semplici non permettono di "tornare indietro". Si può ovviare a questo problema utilizzando due cursori: uno che scandisce la lista e l'altro che viene posizionato ad ogni passo sulla cella precedente a quella di scansione.

Assumiamo ad esempio di effettuare un inserimento in una lista ordinata, richiedendo che il nuovo elemento venga inserito immediatamente prima del primo elemento maggiore di esso, oppure alla fine se il nuovo elemento risulta maggiore di tutti gli altri. L'inserimento si può effettuare col codice seguente:

```
cell * aux = new cell; // cella per il nuovo elemento
aux->info = x;
cell * cur = l; // cursore elemento corrente, in testa
cell * prev; // cursore el.to precedente, indefinito
while (cur!=nullptr && cur->info <= x)
{                   // avanza entrambi i cursori
  prev = cur;
  cur = cur->next;
}
aux->next = cur;    // aggancia la cella seguente
if (cur == l)
  l = aux;          // inserimento in testa
else
  prev->next = aux; // inserimento in mezzo
```

Notiamo l'uso del corto circuito nel test del ciclo while: se `cur` è nullo, la valutazione di `cur->info` non è valida e non viene valutata. Provare per esercizio a disegnare l'evoluzione della memoria col formalismo

grafico, analogamente alla Figura 6.2 - verificare che il codice funziona in tutti i casi seguenti: inserimento in una lista vuota, inserimento in fondo, inserimento in mezzo, inserimento in testa.

La cancellazione di un elemento è simile all'inserimento: si cicla fino a trovare l'alimento da cancellare, poi si aggancia la cella che lo precede a quella che lo segue e infine si restituisce allo heap la cella dell'elemento da cancellare. Anche in questo caso è necessario mantenere il cursore alla cella precedente. Vediamo il codice:

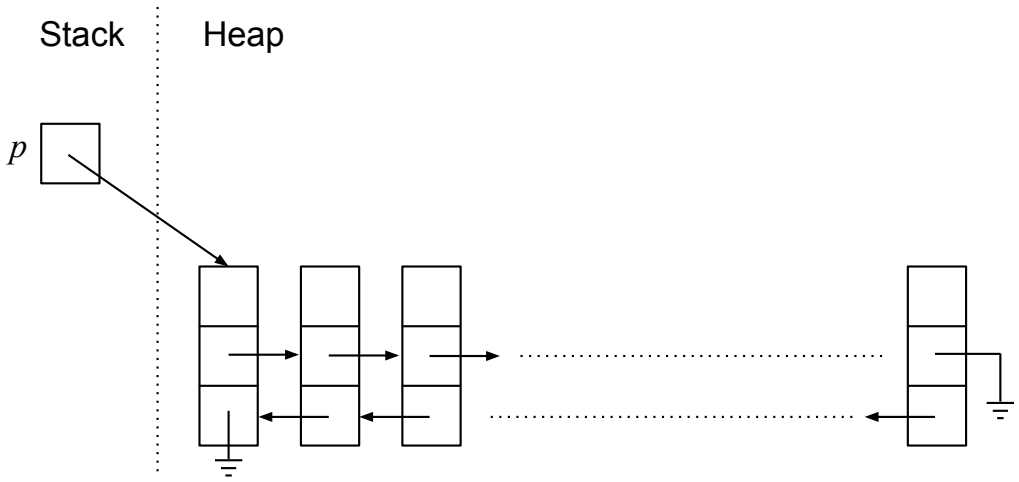
```
cell * cur = l;
cell * prev;
while (cur!=nullptr && cur->info != x)
{
  prev = cur;
  cur = cur->next;
}
if (cur != nullptr)        // altrimenti non si fa niente
{
   if (cur == l)
     l = l->next;            // cancellazione in testa
   else
    prev->next = cur->next;  // cancellazione in mezzo
   delete cur;       // restituisce la cella allo heap
}
```

Con gli ingredienti visti fin qui è possibile realizzare parecchi algoritmi che lavorano su liste. Alcuni possibili esercizi per guadagnare dimestichezza sulla manipolazione di liste e puntatori sono: produrre la copia di una lista senza distruggere quella in input; capovolgere l'ordine di una lista riutilizzando le stesse celle; effettuare l'immersione di due liste ordinate, ossia date due liste ordinate costruire una lista completamente ordinata che contiene gli elementi di entrambe, duplicati inclusi o esclusi, riutilizzando le stesse celle; date due liste ordinate costruire una lista che contiene solo gli elementi che appartengono ad entrambe (intersezione) senza modificare le liste in input. Vedremo nel Capitolo **??** che in alcuni casi risulta più agevole risolvere problemi sulle liste utilizzando tecniche di programmazione ricorsiva, che saranno presentate nel capitolo seguente.

6.6 Approfondimenti sulle liste

6.6.1 Liste doppie

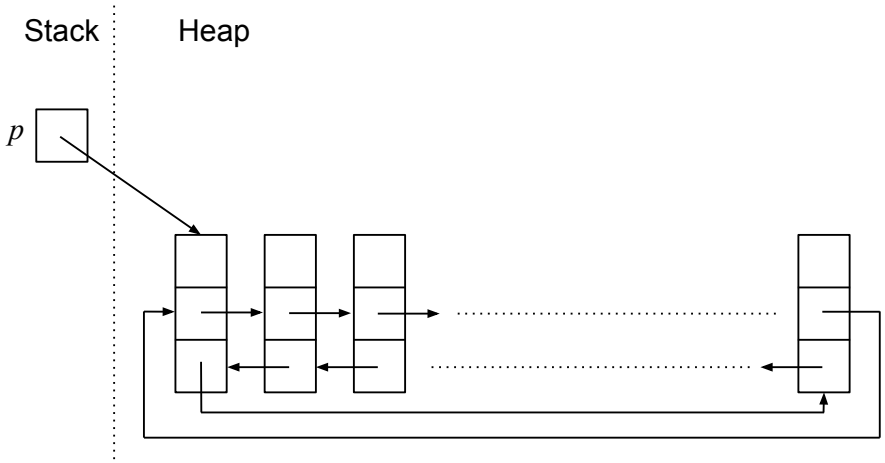
Le liste doppie differiscono da quelle semplici perché ogni cella contiene anche il puntatore alla cella precedente. Quindi una lista doppia ha lo schema seguente:



Questa struttura è ovviamente più dispendiosa in termini di memoria, ma permette di "navigare" agevolmente lungo la lista in entrambe le direzioni. Oltre ad essere necessaria tutte le volte che il programma richieda la scansione bidirezionale, questa struttura semplifica alcune delle operazioni viste in precedenza. Ad esempio, per realizzare inserimento e cancellazione di elementi è sufficiente utilizzare un solo cursore, poiché da una data posizione è possibile accedere sia alla cella successiva che a quella precedente. Tuttavia, nel realizzare operazioni sulle liste doppie è necessaria qualche attenzione in più nell'aggiornamento dei puntatori che legano tra loro le celle. Provare per esercizio a realizzare le operazioni viste nella sezione precedente sulle liste doppie.

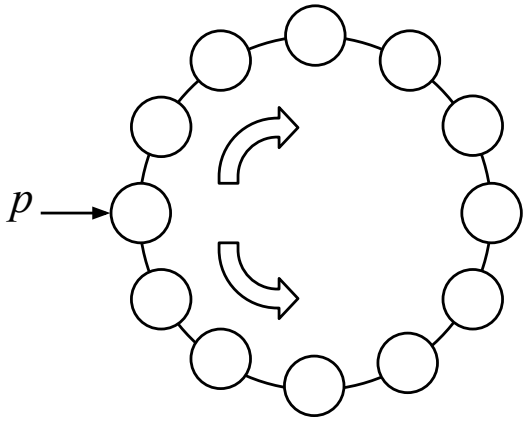
6.6.2 Liste circolari

Le liste circolari, che possono essere sia semplici che doppie, sono contraddistinte dal fatto che l'ultima cella è collegata alla prima, come nello schema seguente (mostrato per liste doppie, quello per liste semplici è analogo).



Ad essere precisi, i termini "prima" e "ultima" cella perdono di significato in questo contesto. La lista è davvero circolare e qualunque punto è buono per entrarci. Il puntatore p infatti serve solo da *entry point* e

può essere fatto scorrere per accedere a punti diversi della lista. La situazione a livello concettuale può essere visualizzata come segue.



Il punto di ingresso sta fermo e la lista è come una ruota libera di girare "a scatti" spostando quindi il punto di ingresso su celle diverse. Con questa struttura non vi è necessità di usare cursori ausiliari perché il punto di ingresso p stesso può essere usato come cursore. Bisogna però fare attenzione durante la scansione a ricordare il punto di partenza per evitare di ciclare all'infinito. Infatti, mancando il "tappo" costituito dal puntatore nullo a fine lista (qui in effetti non c'è una "fine") serve un modo alternativo per capire quando la scansione è terminata. O si tiene in una variabile ausiliaria il numero di celle della lista, oppure si cicla con un cursore che si arresta quando torna sul punto di partenza indicato da p.

Provare per esercizio a definire i tipi necessari per la realizzazione di una lista circolare semplice o doppia e ad implementare semplici funzioni di accesso come ad esempio l'inserimento prima/dopo la cella corrente e la cancellazione della cella corrente.

Parte 7

Ricorsione

La ricorsione è un principio di programmazione molto potente, che sta alla base di alcuni paradigmi di programmazione, come quello *funzionale* o quello *logico* (che verranno esplorati in corsi degli anni successivi). La ricorsione è disponibile anche nella maggior parte dei linguaggi di tipo imperativo, tra cui il C e il C++. La comprensione piena di questo meccanismo non è banale ed il suo uso pratico va amministrato con attenzione.

Secondo una definizione semplicistica, una *funzione ricorsiva* è una funzione "definita in termini di sé stessa". Per altro, tale definizione non risulta affatto intuitiva, in quanto sembra avvolgersi in una spirale infinita (che è proprio ciò che succede se la ricorsione non viene usata nel modo corretto). Un approccio meno semplicistico, ma più chiaro, lega la ricorsione al principio di induzione aritmetica. Nel seguito utilizzeremo questo approccio per definire la ricorsione in modo formale e per vederne alcune semplici applicazioni.

Nella soluzione di problemi, l'esempio più classico di ricorsione può essere riassunto come segue:

- dato un problema P che deve operare su un input I di cardinalità n;
- ammettiamo di saper risolvere P in modo "diretto" se n è piccolo (ossia: esiste una soluzione "semplice" quando i dati sono pochi);
- ammettiamo di saper suddividere l'input I in sottoparti tali che, se disponessimo delle soluzioni per P su ciascuna sottoparte, allora sapremmo combinarle per ricavare una soluzione per P sull'intero input I;
- allora è possibile risolvere P su I con il seguente algoritmo AR:

  Algorithm AR (I) // n è la cardinalità di I
  if n è sufficientemente piccolo **then**
    risolvi P direttamente
  else
    suddividi I;
    risolvi ogni istanza di P su una sottoparte di I, mediante lo stesso algoritmo AR;
    componi le varie soluzioni per ottenere la soluzione su tutto I.

end if

Notiamo che l'algoritmo AR utilizza al suo interno quattro "parti" (che possiamo considerare estensioni procedurali, ma che non sempre vengono realizzate come tali, possono infatti essere semplici blocchi di codice all'interno della stessa funzione che implementa l'algoritmo):

1. la soluzione "diretta" per il caso semplice;
2. il criterio per suddividere I;
3. lo stesso algoritmo AR, utilizzato su una porzione dei dati più piccola rispetto a quella ricevuta in input;
4. l'algoritmo per comporre le soluzioni sulle sottoparti di I.

La/e chiamata/e all'algoritmo AR all'interno del corpo stesso di AR sono dette *chiamate ricorsive.* Mentre per le estensioni 1, 2 e 4 è necessario caso per caso (secondo il problema) sapere come procedere ed utilizzare algoritmi appositi, le chiamate ricorsive sono tipiche di questo schema algoritmico. L'idea intuitiva alla base della ricorsione è che, poichè AR viene chiamato di volta in volta con input sempre più piccoli, a un certo punto ogni chiamata convergerà verso un input sufficientemente piccolo da fermare la ricorsione, calcolando la soluzione in modo diretto. Molti esempi di algoritmi ricorsivi - ma non tutti - possono essere modellati a partire da questo schema.

La ricorsione è molto potente e spesso molto comoda per definire soluzioni semplici ed eleganti, anche per problemi apparentemente complicati, ma va utilizzata con attenzione (vedere la discussione in Sezione 7.6):

- la ricorsione non è strettamente necessaria: ogni algoritmo ricorsivo può essere riscritto in forma iterativa, senza utilizzare la ricorsione, anche se a volte questo può risultare piuttosto difficile;
- la ricorsione può risultare computazionalmente pesante, sia in termini asintotici, che in termini pratici. Infatti, per alcuni problemi esistono soluzioni ricorsive semplici da formulare che però risultano estremamente inefficienti. Per altri problemi esistono soluzioni ricorsive teoricamente efficienti, che però implicano un numero elevato di chiamate a funzione e di allocazione di memoria sullo stack, che possono comprometterne la resa pratica.

Attraverso la ricorsione troveremo soluzioni molto naturali, semplici ed efficienti per manipolare strutture dati come le liste (viste in questo corso) o gli alberi e i grafi (che si vedranno nel corso di Algoritmi e strutture dati) o per risolvere problemi quali il *parsing* di espressioni su un linguaggio del prim'ordine (cfr. anche quanto visto nel corso di EML), ma bisognerà fare attenzione a non abusarne.

7.1 Induzione Aritmetica

Il principio di induzione aritmetica (già visto nell'ambito del corso di EML) è uno strumento matematico che si presta sia a definire insiemi o proprietà, che a fare dimostrazioni. Il dominio su cui una definizione

o dimostrazione induttiva si applica è necessariamente numerabile e totalmente ordinato, quindi assimilabile all'insieme dei numeri interi. Intuitivamente, l'induzione consta di due parti:

- la *base,* in cui si tratta con un numero "piccolo" o, visto in termini informatici, con un piccolo insieme di dati;
- il *passo,* in cui assumendo nota la definizione/dimostrazione su un generico numero (rispettivamente, un generico insieme di dati) si tratta con un numero maggiore (rispettivamente, un insieme di dati più numeroso).

Nella soluzione di problemi, la base sarà direttamente collegata alla soluzione "diretta", mentre il passo sarà naturalmente collegato ad una soluzione ricorsiva. In queste dispense diamo solo alcuni cenni sull'induzione a passi e sull'induzione generalizzata, utili alla comprensione della ricorsione.

7.1.1 Principio di induzione aritmetica "a passi"

Il principio di induzione a passi rappresenta la forma più semplice e del principio di induzione. Sia P un'affermazione sui numeri interi e c un intero. Se valgono le clausole seguenti:

- *base:* P è vera per il numero c
- *passo:* per ogni intero $n \geq c$: se P è vera per n, allora P è vera per $n + 1$

allora P è vera per ogni numero intero $k \geq c$.

Esistono formulazioni alternative, ma del tutto equivalenti, del principio di induzione:

- usando funzioni: invece di una proprietà P sugli interi si parla di una funzione $f_P : \mathbb{Z} \to Bool$, con la convenzione che P vale su k se e solo se vale $f_P(k)$.
- usando insiemi: invece di una proprietà P sugli interi si parla del sottoinsieme $S_P \subseteq \mathbb{Z}$, con la convenzione che P vale su k se e solo se $k \in S_P$.

Di fatto, tutte e tre le formulazioni sono usate, quindi è utile conoscerle e saper passare da una all'altra.

7.1.2 Dimostrazioni per induzione aritmetica "a passi"

Sul principio di induzione aritmetica "a passi" si fondano le dimostrazioni che hanno lo schema seguente. Voglio dimostrare:

Tesi: tutti gli interi $k \geq c$ hanno la proprietà P

Allora :

- *base:* dimostro che c ha la proprietà P;
- *passo:* supponendo che $n \geq c$ abbia la proprietà P (questa viene detta ipotesi induttiva) dimostro che anche $n + 1$ ha la proprietà P.

Il principio di induzione afferma che se sono vere sia la base che il passo, allora è vera la tesi.

7.1.3 Induzione aritmetica "generalizzata"

Diamo l'enunciato del principio utilizzando la definizione basata su proprietà dei numeri interi. Le altre formulazioni e lo schema di dimostrazione si ricavano per analogia con quanto visto sopra (un utile esercizio). Consideriamo quindi una proprità P sugli interi ed un intero c. Se:

- *base:* P vale su c
- *passo:* per ogni $n > c$: se assumendo che P valga su i, per ogni i t.c. $c \leq i < n$ (ipotesi induttiva) allora P vale su n

allora P vale per ogni numero intero $k \geq c$.

Si noti che l'ipotesi induttiva fatta in questo caso costituisce l'unica differenza rispetto al caso precedente. Qui l'ipotesi induttiva è più forte. Infatti, mentre nell'induzione a passi si ipotizza la validità della proprietà sul solo elemento precedente a quello considerato, nell'induzione generalizzata si ipotizza la validità della proprietà su *tutti* gli elementi precedenti, dalla base in poi (tra cui, in particolare, l'elemento immediatamente precedente). Da questo punto di vista, l'induzione generalizzata estende quella a passi: ogni dimostrazione per induzione a passi è anche una dimostrazione per induzione generalizzata, mentre non è vero il contrario.

7.1.4 Esempi

Il fattoriale ed i coefficienti binomiali sono noti dal calcolo combinatorio (cfr. corso di EML). Per entrambi sono ammissibili definizioni e dimostrazioni induttive.

Fattoriale

Se consideriamo n oggetti (distinti), con $n > 0$, e contiamo quante sono le possibili permutazioni di tali oggetti (cioè le possibili successioni di lunghezza n di tali oggetti), vediamo che esse sono date dal prodotto

$n(n-1)(n-2)\ldots 1$. Questo fatto si dimostra facilmente per induzione aritmetica a passi su n (per esercizio). Il numero $n(n-1)(n-2)\ldots 1$ si indica con $n!$ e si legge *n fattoriale.* Per convenzione si pone anche $0! \triangleq 1$. (Il simbolo $\triangleq$ si legge "è uguale per definizione a"). Il fattoriale si può anche definire in modo induttivo come segue:

- *base:* $0! \triangleq 1$
- *passo:* per $n > 0$, $n! \triangleq n \cdot (n-1)!$

La coincidenza delle due definizioni segue banalmente dall'associatività del prodotto tra numeri interi. Si noti come abbiamo usato la tecnica induttiva per fornire una definizione: abbiamo definito il fattoriale dapprima nel caso semplice (la base, sul valore 0) e quindi siamo ricorsi al passo induttivo per definirlo nel caso generale.

Coefficienti binomiali

La definizione standard dei coefficienti binomiali usa il fattoriale. Siano n e k interi tali che $0 \leq k \leq n$. Allora si definisce:

$$\left(\begin{array}{c} n \\ k \end{array} \right) \triangleq \frac{n!}{k!(n-k)!}.$$

Vi sono problemi nell'utilizzare questa definizione per scrivere un algoritmo che calcola il coefficiente binomiale. Infatti, i termini fattoriali possono avere un valore molto grande, che eccede le dimensioni massime per la rappresentazione degli interi, anche quando il valore del coefficiente binomiale è piccolo. Un caso limite è rappresentato dal coefficiente

$$\left(\begin{array}{c} n \\ n \end{array} \right) = 1$$

in cui il valore di n può essere arbitrariamente grande e quindi il suo fattoriale $n!$ può diventare facilmente troppo grande per essere rappresentato, mentre il risultato finale è un numero decisamente piccolo. Questo problema si può aggirare utilizzando una definizione induttiva dei coefficienti binomiali. Partendo dalla definizione basata sul fattoriale, è facile verificare, per sostituzione, che la seguente definizione induttiva coincide con quella precedente:

- *base:* $\left(\begin{array}{c} n \\ 0 \end{array} \right) \triangleq \left(\begin{array}{c} n \\ n \end{array} \right) \triangleq 1$
- *passo:* per $0 < k < n$, $\left(\begin{array}{c} n \\ k \end{array} \right) \triangleq \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ k \end{array} \right)$.

Anche in questo caso, abbiamo definito i coefficienti binomiali dapprima in casi semplici (quando $k = 0$ e quando $k = n$) e poi, ricorrendo al passo induttivo, li abbiamo definiti nel caso generale. Questa seconda definizione ci fornisce la base per un algoritmo ricorsivo, che sfrutta il passo per espandere il calcolo di ogni

coefficiente binomiale, rifacendosi al calcolo di uno più semplice ed evitando sistematicamente il calcolo dei fattoriali (che infatti sono spariti dalla formula). In effetti, tramite questo procedimento non si effettuano neppure moltiplicazioni, ma solo somme. L'algoritmo che ne deriva (che vediamo nella prossima sezione) è molto inefficiente, perché calcola molte volte le stesse cose, ma almeno evita di incorrere nei problemi numerici di un algoritmo diretto basato sul calcolo del fattoriale.

7.2 Funzioni ricorsive

Come già anticipato, gli esempi più semplici di funzioni ricorsive possono essere derivati direttamente dalle definizioni induttive. Vediamo di seguito alcuni esempi che si rifanno alle definizioni date nella sezione precedente.

7.2.1 Fattoriale

Consideriamo il fattoriale, come definito in precedenza. È banale scrivere una funzione iterativa che lo calcola:

function fatt(int n) : int
$aux = 1$
for $i = 1$ to n **do**
  $aux = aux * i$
end for
return aux

Ma possiamo anche scrivere la seguente funzione ricorsiva:

function fr(int n) : int
if $n < 1$ **then**
  return 1
else
  return $n*$ fr$(n-1)$
end if

Si noti come la funzione ricorsiva derivi direttamente dalla definizione induttiva. Il fattoriale ci fornisce un esempio di ricorsione diretta: nel corpo della funzione fr compare una chiamata a fr. Dal punto di vista del linguaggio, il punto chiave è che il nome fr è noto all'interno del corpo della funzione e si riferisce alla funzione stessa.

Una chiamata a funzione ricorsiva genera una *gerarchia di chiamate,* poichè la funzione invoca ricorsivamente un'altra istanza di sé stessa, che a sua volta può invocarne un'ulteriore istanza, e così via. Ogni chiamata

nella gerarchia non potrà essere conclusa, quindi resterà in sospeso, finché non terminano tutte le chiamate sottostanti.

La funzione fr è un esempio di *ricorsione in coda,* ossia di funzione in cui l'unica chiamata ricorsiva compare come ultima istruzione della funzione stessa. Gli algoritmi che utilizzano la sola ricorsione in coda sono facilmente sviluppabili in forma iterativa, come appunto avviene nella versione fatt del calcolo del fattoriale. Lo "srotolamento" (*unroll*) della gerarchia di chiamate di una funzione con ricorsione in coda è particolarmente semplice. Per capire come procede l'esecuzione di una chiamata alla funzione fr, supponiamo che in un programma ci sia l'istruzione **print**(fr(3)). L'esecuzione procede schematicamente così:

$$\text{fr(3)} \rightarrow 3 * \text{fr(2)} \rightarrow 3 * 2 * \text{fr(1)} \rightarrow 3 * 2 * 1 * \text{fr(0)} \rightarrow 3 * 2 * 1 * 1 \rightarrow \textbf{print}\ 6.$$

A basso livello, la gerarchia di chiamate viene gestita nel modo presentato nel Capitolo 4, ossia: si salvano sullo stack lo stato della funzione chiamante e i parametri attuali per la funzione chiamata prima di invocare quest'ultima; lo stack viene ripristinato alla situazione precedente alla fine di ogni chiamata, restituendo al chiamante il risultato (eventuale) generato dalla funzione chiamata. È facile intuire come le funzioni ricorsive possano risultare particolarmente dispendiose, in termini di memoria stack, se la gerarchia di chiamate ricorsive si allunga molto. Nel caso di fr(n) avremo esattamente n chiamate ricorsive annidate. In altri algoritmi questo numero può crescere molto (cfr. esempio seguente) mentre in altri ancora può essere piuttosto piccolo (cfr. ricerca binaria).

7.2.2 Coefficienti binomiali

Un altro esempio naturale di funzione ricorsiva viene dai coefficienti binomiali. Come già notato, un'implementazione diretta tramite calcolo dei fattoriali è molto semplice ma condurrà facilmente a problemi derivanti dai limiti di rappresentazione dei numeri interi. La funzione:

function cbin(int n, int k) : int
return fatt(n)/(fatt(k)*fatt($n-k$))

può facilmente generare un run-time error, ad esempio invocando cbin(1000,1000), anche se il risultato dovrebbe essere 1. Si noti che questo è del tutto indipendente dal fatto che la funzione per il calcolo del fattoriale sia stata implementata in modo iterativo o in modo ricorsivo: in entrambi i casi il fattoriale di 1000 è un numero troppo grande per poter essere rappresentato. Vediamo invece come si può procedere applicando la definizione induttiva:

function cbr(int n, int k) : int
if ($k < 0$ or $n < k$) **then**
  return -1 // codice di errore
else if ($k = 0$ or $n = k$) **then**
  return 1
else
  return cbr($n-1$,$k-1$)+cbr($n-1$,k)
end if

Anche in questo caso l'algoritmo ricorsivo si ottiene come traduzione diretta della definizione induttiva e le chiamate ricorsive compaiono come ultime istruzioni, ma non si tratta di ricorsione in coda. Come si può notare, infatti, le chiamate ricorsive all'interno dell'algoritmo sono due e portano ad uno sviluppo piuttosto complicato, che rende l'algoritmo inefficiente. In effetti, si noti come il numero in output venga costruito sommando un'unità alla volta, unità che vengono restituite dalle chiamate ricorsive che rientrano nel caso base, mentre le chiamate dei livelli superiori si limitano ciascuna a fare una somma. Pertanto, assumendo che il valore calcolato alla fine sia m, ci saranno in totale m chiamate ricorsive nel caso base (quelle che forniscono tutte le unità da sommare). È facile verificare che ci saranno inoltre $m-1$ chiamate ricorsive che generano a loro volta altre chiamate ricorsive, per un totale di $2m-1$ chiamate. Poichè m può essere un numero parecchio elevato (può essere un esponenziale in n, tanto più grande, a parità di n, quanto più k si avvicina al valore $n/2$) le chiamate ricorsive possono essere davvero tante. Per altro, la profondità della gerarchia di chiamate non sarà mai molto elevata, essendo limitata da n come nel caso del fattoriale. Queste cose saranno chiarite meglio più avanti, quando impareremo a valutare la complessità degli algoritmi ricorsivi.

7.3 Ricorsione in coda

Lo schema più semplice di algoritmo ricorsivo si realizza mediante la cosiddetta ricorsione in coda. In questo caso, l'input viene concettualmente diviso in due parti: il primo elemento (o comunque, un elemento scelto dall'input secondo qualche criterio) e "tutto il resto". Il problema in oggetto viene risolto in modo diretto sull'elemento che è stato isolato dagli altri, mentre la soluzione del problema su "tutto il resto" viene demandata ad una chiamata ricorsiva. Il calcolo del fattoriale visto in precedenza fornisce un esempio di ricorsione in coda, ma questo schema si applica in molti altri casi. Spesso la ricorsione in coda rappresenta un'alternativa rispetto a tecniche iterative, che può facilitare la scrittura del codice anche se a volte ne può diminuire l'efficienza.

7.3.1 Liste e ricorsione

Le liste sono strutture dati che ben si prestano ad essere trattate con tecniche ricorsive e con la ricorsione in coda in particolare. Questo approccio ad esempio è fondativi per il linguaggio Lisp, che è un linguaggio (non imperativo) in cui liste costituiscono la struttura dati principale e il meccanismo di calcolo soggiacente è

inerentemente ricorsivo. Vediamo di seguito come alcune delle operazioni su liste viste nel capitolo precedente, ed altre ancora, possono essere implementate facilmente mediante la ricorsione in coda.

Come primo esempio consideriamo la scansione sequenziale, ad esempio per trovare se un elemento appartiene o meno ad una lista. La logica con cui viene progettato l'algoritmo iterativo è: *scandisci la lista finché non trovi l'elemento cercato, oppure la sua fine*. L'algoritmo ricorsivo usa invece la logica seguente: *se la lista è vuota allora non contiene l'elemento cercato; se l'elemento cercato corrisponde al primo della lista allora lo contiene; altrimenti la risposta si trova cercando (ricorsivamente) sul resto della lista*. La lista viene scandita nello stesso ordine in entrambi i casi, ma come si vede la logica sottostante è diversa. Anche in pratica le cose vanno in modo diverso: mentre nel caso iterativo la scansione avviene attraverso un ciclo spostando un puntatore (cursore) mantenuto in una variabile locale, nell'altro caso, attraverso la ricorsione in coda, si scatena una cascata di chiamate ricorsive profonda quanto la lunghezza della lista stessa. Il codice di una tale funzione di ricerca potrebbe essere pertanto implementato nel modo seguente:

```
bool member(list l, T x)
{
   if (l == nullptr) return false;     // lista vuota
   if (l->info == x) return true;  // trovato in testa
   return member(l->next,x);       // cerca nel resto
}
```

Qualora il tipo base `T` ammetta un ordinamento totale e la lista sia ordinata, è facile modificare il codice con un ulteriore condizionale per terminare la ricerca con esito negativo qualora si incontri un elemento maggiore di `x`. Sempre nel caso di liste ordinate, l'inserimento di un elemento al posto giusto può essere realizzato con un meccanismo ricorsivo, più facilmente di quanto visto in precedenza col meccanismo iterativo. La logica seguita dall'algoritmo ricorsivo è simile a quella vista per la scansione: *se la lista è vuota oppure se il primo elemento della lista è maggiore di quello da inserire allora inserisci il nuovo elemento in testa; se invece il primo elemento della lista è uguale a quello da inserire allora non fare niente (stiamo infatti assumendo di non voler inserire duplicati); altrimenti inserisci ricorsivamente l'elemento nel resto della lista*. Il codice corrispondente illustra un esempio di implementazione sulle liste semplici:

```
void insert(list &l, T x)
{
   if ((l == nullptr) || (l->info >x))
   {                        // inserisci in testa
        cell * aux = new cell;
        aux->info = x;
        aux->next = l;
        l = aux;
   }
   else if (l->info == x)
      return;               // elemento gia' presente
```

```
   else
      insert(l->next,x);   // inserisci nel resto
}
```

Notiamo qui come il fatto che la lista sia passata per riferimento permetta di aggiornare correttamente i legami tra gli elementi della lista in conseguenza dell'inserimento. Nel codice l'inserimento della nuova cella viene sempre trattato come un inserimento in testa. Tuttavia, questo avviene in testa *all'intera lista* solo nel caso in cui sia eseguito nella *prima* chiamata, cioè solo se la lista è vuota o il nuovo elemento è minore di tutti quelli presenti. In tutti gli altri casi, l'inserimento avviene in una *chiamata ricorsiva* e quindi in testa a una *sottolista* che costituisce il *resto* della lista globale da un certo punto in poi. Si noti che in quest'ultimo caso, la testa di tale sottolista altro non è che il campo `next` della cella che la precede (ossia il parametro attuale della chiamata ricorsiva, passato per riferimento) che pertanto viene aggiornato correttamente.

Con un meccanismo del tutto analogo è molto facile realizzare la cancellazione di un elemento. In questo caso, la logica è: *se la lista è vuota, non c'è niente da cancellare; altrimenti se l'elemento da cancellare è il primo della lista, toglilo dalla testa; altrimenti cancellalo ricorsivamente dal resto della lista.* Come si vede, l'unica operazione da realizzare in modo diretto è la cancellazione di un elemento dalla testa della lista, che si implementa facilmente (per esercizio), mentre la cancellazione degli elementi "interni" viene gestita dalla ricorsione.

Un esempio ancor più interessante è quello del capovolgimento di una lista. In questo caso, si prende in input una lista e si vuole ottenere come risultato una lista formata dagli stessi elementi ma elencati in ordine inverso. Se la lista in input deve essere mantenuta invariata, mentre quella in output sarà una nuova lista (contenente copie degli elementi in input), la funzione può essere realizzata molto facilmente, sia con un meccanismo iterativo che ricorsivo, con la logica seguente: *inizializza in output una lista vuota; scandisci la lista in input e per ogni elemento presente fai un inserimento dello stesso valore in testa alla lista in output.* Provare per esercizio. Tuttavia, se la lista in input deve essere "demolita" per costruire quella in output riutilizzando le stesse celle (quindi senza fare una copia degli elementi), la realizzazione di una funzione basata sul meccanismo iterativo è tutt'altro che semplice, mentre quella basata sulla ricorsione si può implementare facilmente con il codice seguente (su un tipo di lista semplice con elementi in `T`):

```
void reverse(list &l)
{
   list l1;
   // se la lista contiene zero o un elemento resta invariata
   if ((l == nullptr) || (l->next == nullptr)) return;
   l1 = l->next;
   reverse(l1);       // capovolgi il resto
   l->next->next = l;
   l->next = nullptr;
   l = l1;
}
```

Proviamo ad analizzare questo codice che, nonostante l'estrema concisione, è tutt'altro che banale. La logica è la seguente: *se la lista è vuota o contiene un solo elemento, la soluzione coincide con la lista stessa (non è necessario capovolgerla, quindi non si fa niente); altrimenti, si capovolge ricorsivamente il resto della lista (cioè la lista tolto il primo elemento) e poi si appende in fondo a tale lista capovolta l'elemento che inizialmente si trovava in testa.* L'istruzione condizionale si incarica del caso base della ricorsione (zero o un elemento). La chiamata ricorsiva si incarica di capovolgere il resto della lista, il cui punto di accesso viene memorizzato nella variabile ausiliaria `l1`. A quel punto, il campo `next` del primo elemento della lista `l` sta ancora puntando all'elemento che in origine era il secondo della lista e che ora è esattamente l'ultimo della lista capovolta `l1`. L'istruzione che segue la chiamata ricorsiva fa quindi in modo che l'elemento puntato da `l` venga appeso come successore dell'ultimo elemento della lista `l1` e l'istruzione successiva fa in modo che tale elemento diventi l'ultimo della lista mettendo il suo campo `next` al valore `nullptr`. L'ultima istruzione infine si incarica di aggiornare la lista in input al nuovo valore.

7.4 Tecniche divide-et-impera

In molti casi, la soluzione di problemi mediante algoritmi ricorsivi deriva in modo naturale dall'applicazione di un paradigma di programmazione noto col nome *divide-et-impera* (*divide-and-conquer* in inglese), che può essere riassunto con uno schema simile a quello descritto all'inizio del capitolo. Dato un problema P che deve operare su un input I di cardinalità n:

1. se n è sufficientemente piccolo, risolvi P in modo diretto;
2. altrimenti:
   (a) suddividi I in sottoparti di cardinalità minore
   (b) risolvi P sulle sottoparti
   (c) combina i risultati ottenuti sulle sottoparti per ottenere il risultato finale su tutto I.

L'implementazione del passo 2.b implica di solito l'uso della ricorsione. Le parti 2.a (divide) e 2.c (impera) possono essere più o meno complicate secondo la natura e problema P e a volte sono del tutto banali: ad esempio la fase 2.c può essere assente quando la soluzione finale derivi direttamente dalla soluzione su una sola delle sottoparti.

Ovviamente non tutti i problemi possono essere risolti con questo paradigma e non tutti gli algoritmi ricorsivi derivano da esso, ma esistono diversi esempi notevoli di problemi risolvibili con il paradigma divide-et-impera per i quali si deriva naturalmente un algoritmo ricorsivo. Nel seguito ne vediamo un paio di carattere fondamentale.

7.4.1 Ricerca binaria

Il problema trattato è quello della ricerca di un elemento in una sequenza ordinata (secondo un criterio di ordinamento noto). Come già visto nel caso delle liste, il problema è sempre risolvibile in modo banale attraverso una *scansione sequenziale,* che viene interrotta in uno dei tre casi seguenti: si incontra l'elemento cercato, si incontra un elemento di valore maggiore di quello cercato (rispetto all'ordinamento); si incontra la fine della sequenza. Questo algoritmo si implementa banalmente in modo iterativo con un ciclo, ma si può anche implementare con una ricorsione in coda come segue:

function scan(sequenza di elem s, elem x) : bool
if is_empty(s) **or** first(s) $> x$ **then**
  return false
else if first(s) $= x$ **then**
  return true
end if
return scan(rest(s), x)

dove la funzione first() restituisce il primo elemento della sequenza, mentre la funzione rest() restituisce il resto della sequenza tolto il primo elemento. L'algoritmo in entrambi i casi (iterativo e ricorsivo) obbliga ad una scansione di tutta la sequenza nel caso peggiore, che si verifica quando x ha un valore maggiore di tutti gli elementi contenuti in s. In questo caso inoltre l'implementazione con ricorsione in coda non presenta alcun vantaggio rispetto a quella iterativa, anzi, appesantisce l'algoritmo sia in termini di tempo di esecuzione che in termini di memoria, per la gestione dello stack.

Se la sequenza ammette l'accesso diretto, quindi se ad esempio è mantenuta in un array o un vector, è invece possibile usare una tecnica alternativa e molto più efficiente, detta di *ricerca binaria*:

Algorithm ricbin(s,x)
if s vuota **then**
  x non trovato
else
  confronta x con l'elemento e che occupa la posizione centrale di s
  if $x = e$ **then**
    x trovato
  else if $x < e$ **then**
    cerca x nella parte di s che precede e
  else
    cerca x nella parte di s che segue e
  end if
end if

La ricerca si conclude (con esito positivo) quando e viene trovato oppure (con esito negativo) quando la parte di sequenza da analizzare risoluta vuota. Questa tecnica risulta molto più efficiente della precedente poiché permette di scartare in blocco intere porzioni di sequenza senza analizzare gli elementi al loro interno. Analizzeremo in seguito la complessità computazionale della ricerca sequenziale e della ricerca binaria e verificheremo che il vantaggio di quest'ultima sulla prima è davvero enorme, specie quando si tratti di sequenze molto lunghe.

La ricerca binaria si può implementare sia con un algoritmo iterativo che con uno ricorsivo (con ricorsione in coda). Entrambi sono molto semplici, tuttavia quello ricorsivo corrisponde ad una traduzione pressoché immediata dello schema appena illustrato.

L'unica accortezza, valida in entrambi i casi riguarda lo strumento con cui delimitare la porzione di se-

quenza di interesse. Vogliamo evitare di copiare quest'ultima su una variabile ausiliaria perché il costo computazionale della copia vanificherebbe completamente i vantaggi del metodo. Volendo indicare una porzione di array (o vector) possiamo utilizzare due indici (interi) che indicano rispettivamente la prima e l'ultima casella di interesse. Tali indici saranno mantenuti in variabili ausiliarie nella versione iterativa e in parametri della funzione nella versione ricorsiva.

La versione iterativa può essere scritta come segue:

```
function ricbin(array di elem s, elem x) : bool
// s ha n elementi numerati da 0 a n − 1
l = 0; r = n − 1
// porzione non vuota sse l ≤ r
while l ≤ r do
  m = (l + r)/2                  // m sta a metà tra l ed r
  if x = s[m] then
    return true
  else if x < s[m] then
    r = m − 1                    // restringe la ricerca alla parte sinistra
  else
    l = m + 1                    // restringe la ricerca alla parte destra
  end if
end while
return false
```

Nell'algoritmo qui sopra, le due variabili locali l ed r vengono utilizzate per per delimitare rispettivamente a sinistra e a destra la porzione di s da analizzare. Nella versione ricorsiva, invece, la funzione ricbin chiama un funzione ricorsiva che riceve come parametri tali limiti l ed r::

```
function ricbin(array of elem s, elem x) : bool
rbin_ric(s, x, 0, n − 1)

function rbin_ric(array di elem s, elem x, int l, int r) : bool
// la porzione di s da analizzare va da l a r
if l > r then
  return false                   // porzione vuota
end if
m = (l + r)/2
if x = s[m] then
  return true
else if x < s[m] then
  return rbin_ric(s, x, l, m − 1)          // restringe la ricerca alla parte sinistra
else
```

return rbin_ric($s, x, m+1, r$) // restringe la ricerca alla parte destra
end if

Questo meccanismo di implementazione di un algoritmo mediante una funzione ricorsiva ausiliaria è molto comune. Infatti, affinché la ricorsione funzioni è necessario che la funzione riceva parametri diversi ad ogni chiamata (nel nostro caso, la porzione di sequenza in cui effettuare la ricerca, delimitata dagli indici l ed r). Il meccanismo ricorsivo pertanto viene implementato nella funzione ausiliaria, mentre quella principale si limita ad invocare la funzione ausiliaria, passandole i parametri corrispondenti all'intero insieme di dati (nel nostro caso, specificando come estremi i valori 0 ed $n-1$).

Per esercizio:

- provare a simulare l'esecuzione della chiamata rbin_ric($s, x, 0, n-1$), con n piccolo e nei due casi: x è in s, x non c'è;
- riscrivere l'algoritmo, in modo che produca l'indice della posizione in cui si è trovato l'elemento (-1 se l'elemento non si trova).

7.4.2 Merge sort

Il problema, già affrontato in precedenza e risolto con algoritmi iterativi non particolarmente efficienti (Insertion sort e Selection sort) è quello di ordinare una sequenza di valori. Il merge sort è un algoritmo ottimale dal punto di vista della complessità asintotica. Non richiede che la sequenza sia ad accesso diretto (si può implementare agevolmente anche sulle liste) e può essere adattato per operare su file (quindi può funzionare anche in memoria secondaria, ma richiede alcune modifiche).

L'idea dell'algoritmo è una realizzazione diretta del principio divide-et-impera: si divide la sequenza "a metà"; si ordinano separatamente le due parti con due chiamate ricorsive alla procedura stessa; poi si combinano le due parti ordinate, con una operazione di "fusione" (*merging*, in inglese).

Prima di tutto, vediamo che l'operazione di fusione è facile e si realizza con un algoritmo iterativo. In generale: date due successioni ordinate $X = x_1, ...x_p$ e $Y = y_1, ..., y_q$ si tratta di ottenere un'unica successione ordinata Z, di lunghezza $p+q$, contenente tutti gli elementi di X e tutti quelli di Y (sono ammessi elementi duplicati). Ad esempio, se

$$X = 2, 3, 10, 20$$

e

$$Y = 4, 15, 25, 27$$

allora

$$Z = 2, 3, 4, 10, 15, 20, 25, 27.$$

L'idea è di sfruttare l'ordinamento preesistente di X e Y per minimizzare il numero di confronti necessari. Si scandiscono X e Y in parallelo, a partire dal primo elemento di entrambe. A questo scopo usiamo due cursori:

i per la sequenza X e j per la sequenza Y. Si noti che i due cursori sono generici: nel caso di array o vettori saranno indici interi, nel caso di liste saranno puntatori. L'elemento puntato in ogni momento da un cursore si dice elemento corrente. Ad ogni passo, si sceglie l'elemento minimo tra i due correnti, lo si copia in Z e si avanza il cursore che si trova su tale elemento minimo. Usiamo il cursore i per la sequenza X, j per la sequenza Y e k per la sequenza Z. Assumendo per il momento di usare array e indici come cursori, con la notazione usuale, l'algoritmo può essere scritto come segue:

$i = 0; j = 0; k = 0;$
while ($i < p$ **and** $j < q$) **do**
  if $X[i] < Y[j]$ **then**
    $Z[k] = X[i]; i++;$
  else
    $Z[k] = Y[j]; i++;$
  end if
  $k++;$
end while
while $i < p$ **do**
  $Z[k] = X[i]; i++; k++;$
end while
while $j < q$ **do**
  $Z[k] = X[j]; j++; k++;$
end while

Il primo ciclo **while** realizza la scansione parallela di X e Y. All'uscita da tale ciclo, una delle due sequenze è terminata e l'altra viene copiata semplicemente in fondo a Z, eseguendo uno dei due cicli **while** successivi. Notare che tali cicli vengono eseguiti in alternativa, in quanto all'uscita dal primo **while** solo una delle due condizioni è verificata.

Nel caso specifico dell'ordinamento, le due sequenze X ed Y corrispondono in realtà a due porzioni consecutive di una stessa sequenza in input, mentre la sequenza Z è una variabile ausiliaria, che verrà poi copiata sulla sequenza originaria, come vedremo.

Torniamo ora allo schema dell'ordinamento sapendo che, più o meno, sappiamo come fare la fusione. L'algoritmo di Merge sort si implementa tramite una funzione ausiliaria, con uno schema analogo a quello visto per la ricerca binaria. La procedura ausiliaria ms è ricorsiva ed opera su una porzione di sequenza, delimitata da due cursori *inf* e sup. La procedura principale mergesort si limita a chiamare la ms sull'intera sequenza.

procedure mergesort(IN-OUT sequenza s)
inf = first(s); sup = last(s);
ms(s,*inf*,sup)

procedure ms(IN-OUT sequenza s, IN cursori *inf*, sup)

if *inf* $\geq$ sup **then**
  return
else
  $med =$ cursore all'elemento di mezzo della porzione da *inf* a sup
  ms(s,*inf*,med) // ordina ricorsivamente la prima metà
  ms(s,next(med), sup) // ordina ricorsivamente la seconda metà
  merge(s,*inf*,med, sup)
end if

Notare che i cursori sono di un tipo generico, potrebbero essere implementati con indici interi su array o vettori e con puntatori su liste. Si prevede solo che siano in grado di scorrere la sequenza tramite primitive first, last e next e che sia definito l'operatore di confronto $<$. Determinare il punto medio della sequenza può essere un'operazione relativamente costosa (as esempio sulle liste, in cui si richiede una scansione) ma questo non compromette la complessità computazionale dell'algoritmo. È invece importante sottolineare l'importanza di passare il paramentro s per riferimento, per evitare di copiare ad ogni chiamata ricorsiva l'intera sequenza in input.

La procedura merge implementa l'algoritmo di immersione visto in precedenza, utilizzando i tre cursori *inf*, med e sup per delimitare le due porzioni di sequenza da immergere tra loro. Assumiamo per il momento di disporre di tale procedura e cerchiamo di capire come funziona ms e se produce il risultato voluto. Questo ci condurrà ad un primo esempio di *prova di correttezza,* ossia la dimostrazione formale che un algoritmo è in grado di produrre il risultato voluto su qualunque input.

Il ragionamento informale è il seguente. Cosa deve fare ms(s,*inf*,sup)? Deve ordinare la porzione (sottosequenza) di s compresa tra *inf* e sup. Sappiamo che *inf*$\leq sup$.

- se *inf*$= sup$ non c'è niente da fare, la sottosequenza è costituita di un unico elemento e quindi già ordinata;
- altrimenti *inf*$< sup$: se suppongo che le chiamate ricorsive ms(s,*inf*,med) e ms(s,next(med),sup) facciano bene il loro lavoro, allora il loro effetto è: le porzioni di s da *inf* a med e da next(med) a sup sono ciascuna ordinata; se inoltre la procedura merge è capace di prendere le due sottosequenze consecutive ordinate e fonderle, allora effettivamente, ms(s,*inf*,sup) ordina s da *inf* a sup.

Questo ragionamento può lasciare perplessi, ma funziona. È anche importante capire perché. Per prima cosa: da quello che abbiamo già visto, dovrebbe essere chiaro che la procedura merge soddisfa le nostre richieste. A questo punto resta solo la supposizione che le due chiamate interne ad ms funzionino. Può sembrare che ci si morda la coda; non è così e lo si vede trasformando questo ragionamento in una dimostrazione per induzione (generalizzata), come segue. Per semplicità, ma senza perdere in generalità, trattiamo s come un array e i cursori come indici interi, considerando che gli elementi di s siano numerati da 0 a $n-1$.

Ipotesi:

- $0 \leq$ *inf* $\leq sup < n$

- la procedura merge soddisfa la seguente specifica: se s è ordinato (in modo crescente) da *inf* a med e da next(med) a sup (estremi compresi), la chiamata merge(s,*inf*,med, sup) ordina la porzione di s che va da *inf* a sup (estremi compresi).

Tesi: la chiamata ms(s,*inf*,sup) ordina s da *inf* a sup.

Dim: per induzione aritmetica generalizzata sulla lunghezza $k = sup - inf + 1$ della sottosequenza.

- base: $k = 0$ oppure $k = 1$. In questi casi la sottosequenza è vuota oppure è costituita di un solo elemento, quindi non c'è nulla da fare; correttamente, ms in questo caso non fa nulla (esegue solo il test dell'**if** ed esce).

- passo: $k > 1$. La procedura ms in questo caso: calcola med, esegue le due chiamate ricorsive e poi chiama merge. Il punto chiave della dimostrazione è che le due chiamate ricorsive sono fatte su parti di s di dimensione strettamente minore di k. Assumiamo, per ipotesi induttiva, che ms produca un risultato corretto su qualunque sottosequenza di lunghezza $< k$. Consideriamo la prima chiamata: ms(s, *inf*, med). Dalla definizione di med abbiamo che $med < sup$, quindi $med - inf + 1 < k$, quindi per ipotesi induttiva ms(s, *inf*, med) ordina la sottosequenza di s da *inf* a med. Analogamente, $med + 1 > inf$, quindi $inf - (med + 1) + 1 < k$ e dunque ms($s, med + 1, sup$) ordina la sottosequenza di s da $med + 1$ a sup. Pertanto merge(s,*inf*,med, sup) riceve un input correttamente ordinato nelle due sottosequenza e completa il lavoro, ordinando s da *inf* a sup. □

Notare il fatto che, nella dimostrazione, è necessario disporre del principio di induzione generalizzata. Non basterebbe il principio di induzione a passi perché l'ipotesi induttiva non si fa su un input minore di un solo elemento, ma su qualunque input di cardinalità minore, essendo le due chiamate ricorsive fatte su input che hanno (circa) la metà degli elementi.

Per concludere, vediamo l'imlementazione della procedura merge. È sufficiente riprendere lo schema di immersione visto all'inizio, utilizzando le porzioni opportune di s al posto delle sequenze X e Y e una sequenza ausiliaria aux al posto di Z. Dopo l'immersione, la sequenza aux sovrascrive la corrispondente sottosequenza di s:

procedure merge(IN-OUT sequenza s, IN cursori *inf*, med, sup)
$i =$ *inf* ; $j = med$; $k = 0$;
while ($i \leq med$ **and** $j \leq sup$) **do**
  if $s_i < s_j$ **then**
    $aux_k = s_i$; $i =$next(i);
  else
    $aux_k = s_j$; $j =$next(j);
  end if
  $k =$next(k);
end while
while $i \leq med$ **do**
  $aux_k = s_i$; $i =$next(i); $k =$next(k);

end while
while $j \leq sup$ **do**
$aux_k = s_j; j =$next$(j); k =$next$(k);$
end while
copia aux sulla porzione di s da *inf* a sup

Per esercizio: simulare l'esecuzione della chiamata ms(s,0,4) su una sequenza di lunghezza 4 a scelta (sceglierla disordinata).

7.5 Ricorsione indiretta e incrociata

Quelli visti sono tutti esempi di ricorsione diretta: una funzione o procedura contiene, nel corpo, una o più chiamate a sé stessa. Si può anche avere una ricorsione *indiretta*: la funzione P chiama direttamente la funzione Q, che chiama direttamente la funzione R, ..., che chiama direttamente P. Un esempio semplice di ricorsione indiretta è la ricorsione *incrociata* (detta anche ricorsione *mutua*): P chiama Q e Q chiama P. Un esempio notevole di tutto questo si ha nel *parsing* e nella valutazione di espressioni in linguaggi del prim'ordine.

7.6 Quando usare la ricorsione

Difficile dare una risposta netta. Prima di tutto, bisogna familiarizzare con la tecnica. Fatto questo, si vede che in molte situazioni l'approccio ricorsivo risulta chiaro e semplice. Ci sono almeno due casi notevoli:

- algoritmi per lavorare su tipi di dato definiti induttivamente, come le liste (cfr. Capitolo **??**) e gli alberi (che si vedranno nel corso di ASD);
- algoritmi che utilizzano la tecnica del divide-et-impera, come la ricerca, binaria, il merge sort e moltissimi altri.

Abbiamo visto che l'idea della ricerca binaria si può facilmente tradurre in stile iterativo (cioè usando istruzioni di ripetizione, while, repeat, for); decisamente meno facile è la formulazione iterativa dell'idea di merge sort. Analogamente, si vedrà che per lavorare sulle liste è abbastanza indifferente usare uno stile iterativo o uno ricorsivo, mentre lavorando su alberi è in generale più facile usare lo stile ricorsivo.

Va detto che esiste una tecnica standard per eliminare la ricorsione, usando uno stack: in pratica si fa "a mano" quello che l'implementazione della ricorsione fa automaticamente. Tuttavia, in molti casi la trasformazione per eliminare la ricorsione è faticosa e produce algoritmi molto meno chiari di quelli ricorsivi.

Una delle obiezioni che si fa alla ricorsione è che il meccanismo può risultare "costoso" in termini di tempo di esecuzione; del resto anche in assenza di ricorsione, il meccanismo "dichiarazione di funzione – chiamata della funzione" ha un certo costo. Questo punto si chiarirà meglio parlando di complessità degli

algoritmi. Qualcosa si può comunque anticipare. Nei linguaggi di programmazione moderni, tra cui il C e il C++, le funzioni sono considerate uno degli strumenti principali per programmare in modo sicuro e flessibile; quindi la questione: procedure sì, procedure no, è risolta con un sì deciso. Gli eventuali "costi", non rilevanti in assoluto, sono più che bilanciati dai vantaggi per il programmatore. Se poi si confronta, in termini di soldi, il costo del lavoro umano con quello del lavoro della macchina, allora diventa evidente che conviene far lavorare la macchina. In questa ottica, tutte le volte che risulta comodo e naturale usare la ricorsione, conviene usarla. Se l'algoritmo che stiamo progettando, una volta diventato programma, deve essere usato molte volte o è una componente critica di un sistema, bisogna valutarne in qualche modo le prestazioni (ad esempio stimandone la complessità). In qualche caso si vede che l'approccio ricorsivo è troppo inefficiente (in realtà, in genere, non è colpa della ricorsione, ma dell'idea su cui si basa l'algoritmo). Un esempio è il calcolo dei coefficienti binomiali: la funzione ricorsiva che abbiamo discusso precedentemente ha una complessità troppo elevata. A questo punto, si cerca di capire cosa non funziona e di rimediare, magari cambiando strategia. Per i coefficienti binomiali, ad esempio, si può arrivare ad un algoritmo (non ricorsivo) molto più efficiente, basato su un paradigma detto della *programmazione dinamica* (che si vedrà nel corso di ASD).