Università degli studi di Cagliari
Corso di Laurea in Informatica

Dispense del corso di

Programmazione 1

Riccardo Scateni

Anno Accademico 2014-2015

Indice

Capitolo 1

Introduzione

1.1 Algoritmi, programmi e computer

Iniziamo con il definire un insieme di termini che devono obbligatoriamente essere di senso condiviso prima di parlare di qualsiasi metodo di risoluzione automatica di problemi con l'uso di un elaboratore elettronico:

Informatica Disciplina scientifica che si occupa dell'informazione e del suo trattamento in maniera **automatica**;

Computer Macchina elettronica **programmabile** che può svolgere diverse funzioni a seconda delle istruzioni assegnate;

Programma Sequenza di istruzioni elementari che un computer è in grado di comprendere ed eseguire;

Programmazione Attività che consiste nell'organizzare istruzioni elementari, direttamente comprensibili dal computer, in strutture complesse (programmi) al fine di svolgere determinati compiti.

La scrittura di programmi per le macchine di calcolo è una delle principali attività che svolge chi si dedica, professionalmente, ad applicare i concetti dell'informatica per la risoluzione di problemi reali.

1.1.1 Gli algoritmi

Quando si parla di trattamento automatico dell'informazione entrano in gioco due componenti fondamentali:

Strumenti logici Procedimenti di elaborazione **algoritmici**, che, una volta trascritti come programmi, diventano la guida al funzionamento della macchina, il cosiddetto *software*;

Oggetti fisici Elementi che si possono toccare (*hardware*), il computer e le sue componenti che sono il semplice strumento che consente di eseguire le operazioni algoritmicamente definite.

La nozione più importante in assoluto che dobbiamo assimilare per poter pensare di risolvere i problemi in maniera automatica è la nozione di **algoritmo**. Ogni procedimento, per poter essere tradotto in un procedimento automatico deve rispettare le regole che definiscono un metodo come algoritmico:

Algoritmo

Insieme **finito** e ordinato di passi **eseguibili** in tempo finito e **non ambigui** (univocamente definiti), che definiscono un processo che **termina**.

L'algoritmo di Euclide

Uno dei più antichi esempi di algoritmo che si conosca è l'algoritmo attribuito a Euclide per calcolare il massimo comun divisore (MCD) fra due numeri x e y:

1. Calcola il resto della divisione di x per y
2. Se il resto è diverso da zero,
   ricomincia dal passo 1 utilizzando come x il valore attuale di y, e come y, il valore del resto,
   altrimenti
   prosegui con il passo successivo
3. Il massimo comun divisore è uguale al valore attuale di y

Tutte le informazioni necessarie per trovare la soluzione del problema sono descritte nell'algoritmo e chiunque sappia comprendere ed eseguire le operazioni che costituiscono l'algoritmo di Euclide, può **effettivamente** calcolare l'MCD.

L'algoritmo di ordinamento di tutti i numeri primi

Proviamo adesso a scrivere il metodo che consenta di ordinare tutti i numeri primi:

1. Crea un elenco di tutti i numeri primi
2. Ordina l'elenco in modo decrescente
3. Preleva il primo elemento dall'elenco risultante

Dovremmo chiederci, sulla base della definizione che abbiamo dato se questo è effettivamente un algoritmo. No, in effetti, non lo è: la prima e la seconda istruzione non sono **effettivamente eseguibili** in quanto richiedono la manipolazione di un numero infinito di elementi e quindi pur essendo descritto da un numero finito di passi, ognuno dei quali è ben interpretabile, non rispetta **tutte** le condizioni necessarie per poter essere definito algoritmo.

1.1.2 Algoritmi e programmi

Programma

È l'espressione di un algoritmo in un linguaggio che l'esecutore è in grado di comprendere senza bisogno di ulteriori spiegazioni.

Mentre un algoritmo è un oggetto **astratto**, concettuale slegato dalla vera e propria modalità di realizzazione del calcolo che definisce, un programma è un'**espressione concreta** dell'algoritmo, la sua trasformazione in un metodo legato ad un particolare esecutore automatico. Lo stesso algoritmo può essere espresso in differenti linguaggi, in base agli esecutori (computer) ai quali è destinato. La scrittura del programma è sempre una fase successiva all'individuazione dell'algoritmo per risolvere un determinato problema.

1.2 Dal linguaggio macchina ai linguaggi ad alto livello

I due più importanti modelli di computazione, sviluppati nella prima metà del Novecento, che ancora influenzano la teoria e la pratica del calcolo sono la "Macchina di Turing" e la "Macchina di Von Neumann".

La macchina di Turing è un modello di computazione ideale la cui elaborazione ha consentito di sviluppare tutta la conseguente teoria della computabilità e calcolabilità. La macchina di Von Neumann è un modello di architettura per un elaboratore tipo che ha ispirato la progettazione di quasi tutti i computer reali costruiti in seguito.,

1.2.1 La macchina di Von Neumann

Nel 1946, John Von Neumann, un fisico ungherese emigrato negli Stati Uniti, collaboratore del progetto Manhattan, suggerì quella che, secondo lui, avrebbe dovuto essere l'architettura ideale di qualsiasi calcolatore elettronico. In questo modello i componenti fondamentali dell'hardware di un elaboratore sono due:

Memoria è lo spazio che contiene sia il programma da eseguire che i dati da esso utilizzati;

Processore è l'esecutore effettivo delle operazioni che opera ripetendo indefinitamente lo stesso ciclo (vd. figura 1.1) di recupero dei dati dalla memoria, loro decodifica e trasformazione dello stato della computazione in conseguenza dell'esecuzione dell'operazione decodificata.

1.2.2 Linguaggio macchina e assembler

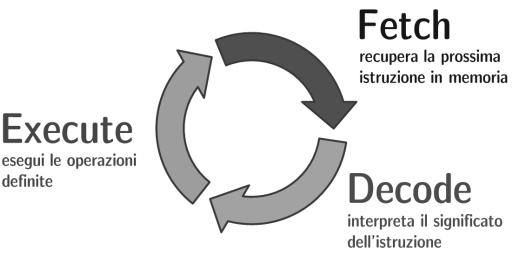


Figura 1.1: Lo schema della macchina di Von Neumann.

Nel passaggio dalla teoria alla realizzazione pratica degli elaboratori elettronici fu quasi scontato che ogni progettista e produttore scegliesse la codifica delle operazioni che più era efficiente dal suo punto di vista e per le competenze ed esigenze che possedevano i progettisti. Questo ha portato a una situazione in cui ogni processore, o famiglia di processori, ha un proprio **linguaggio macchina** con un proprio formato delle istruzioni. Le istruzioni sono **sequenze di bit** che codificano l'operazione da eseguire e gli operandi su cui tale operazione deve essere eseguita (registri, locazioni di memoria, costanti ...). Anche se esiste una sovrapposizione dal punto di vista logico per quanto riguarda le operazioni svolte dai diversi linguaggi delle diverse macchine, ciò non toglie che i linguaggi siano diversi. È quello che succede anche nei linguaggi naturali: in tutte le lingue si ha una parola che indica l'acqua, ma da una lingua all'altra la codifica per esprimere questo concetto cambia.

Il linguaggio **assembler** di un processore non è altro che la **versione simbolica** della codifica ovvero del linguaggio macchina. In Assembler ogni singola istruzione viene identificata con un nome mnemonico (es. `ADD` per indicare l'addizione di due numeri) anziché con il codice numerico effettivamente utilizzato come codice all'interno della macchina.

Un assembler "giocattolo"

Potremmo provare a definire un mini linguaggio Assembler che contenga le istruzioni strettamente indispensabili per compiere le operazioni aritmetiche elementari e per gestire il trasferimento di dati da e per la memoria. Di seguito alcuni esempi di istruzioni, con la relativa descrizione.

Istruzioni per il trasferimento dei dati

LOAD R, x[a]
STORE R, y[b]

[a] carica il valore contenuto nella cella di memoria `x` nel registro `R`
[b] scrivi il valore contenuto nel registro `R` nella cella di memoria `x`

Istruzioni aritmetiche (e logiche)

ADD R1, R2[a]
MUL R1, R2[b]

[a] somma i valori contenuti nei registri `R1` e `R2` e scrivi il risultato in `R1`
[b] moltiplica i valori contenuti nei registri `R1` e `R2` e scrivi il risultato in `R1`

Istruzioni di controllo e di salto

JUMP alfa[a]
JZERO R1, beta[b]

[a] salto *incondizionato* alla riga con etichetta `alfa`
[b] salto *condizionato* alla riga con etichetta `beta`

L'algoritmo di Euclide

Proviamo a tradurre l'algoritmo di Euclide definito in 1.1.1 utilizzando l'Assembler appena definito:

```
      LOAD  R1, 101
      LOAD  R2, 102
rem:  DIV   R1, R2
      MUL   R2, R1
      LOAD  R1, 101
      SUB   R2, R1
      JZERO R2, fine
      LOAD  R1, 102
      STORE R1, 101
      STORE R2, 102
      JUMP  rem
fine: LOAD  R1, 102
      STORE R1, 103
```

Abbiamo utilizzato anche le istruzioni **SUB** e **DIV** per la sottrazione e la divisione di due interi. Si può comprendere dall'esempio come la traduzione dell'algoritmo nel programma, scritto nel linguaggio Assembler definito, ne

diminuisca di molto la capacità di comprensione. L'interpretazione del codice Assembler, quando era l'unica maniera per scrivere programmi per computer, era un'attività molto complessa e impegnativa, riservata, infatti, a pochi esperti.

Se vogliamo elencare gli svantaggi dell'utilizzo del linguaggio macchina (o Assembler) per lo sviluppo di programmi abbiamo:

- È necessario **conoscere i dettagli dell'architettura** del processore utilizzato e il relativo linguaggio;
- Risulta pressoché **impossibile trasportare i programmi** da una macchina ad una differente;
- Il programmatore si specializza nell'uso di **"trucchi" legati alle caratteristiche specifiche della macchina** e, di conseguenza, i programmi risultano difficili da comprendere e da modificare;
- La **struttura logica del programma è nascosta** rendendo il programma stesso difficile da comprendere e da correggere in presenza di errori.

Tutti questi motivi furono la spinta per progettare e realizzare linguaggi indipendenti dall'architettura reale della macchina, quindi, a livello più alto rispetto alle specifiche tecniche dello strumento di calcolo e più vicini allo schema logico degli algoritmi da codificare.

1.2.3 I linguaggi ad alto livello

I linguaggi ad alto livello nascono con l'obiettivo di rendere la programmazione indipendente dalle caratteristiche peculiari della macchina utilizzata. Per questi motivi non sono pensati per essere compresi direttamente da macchine reali ma da macchine **"astratte"**, in grado di effettuare operazioni più ad alto livello rispetto alle operazioni elementari dei processori reali. L'attività di programmazione viene così svincolata dalla conoscenza dei dettagli architetturali della macchina utilizzata.

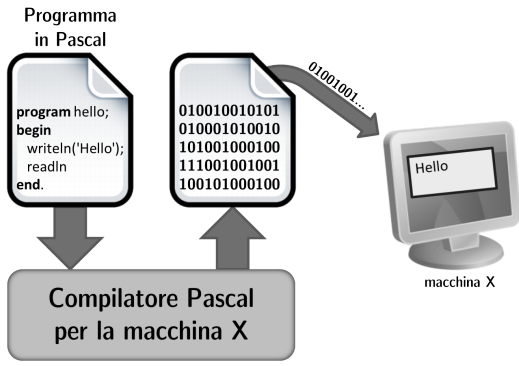


Figura 1.2: Rappresentazione schematica delle funzioni di un compilatore, questo, in particolare, per il linguaggio Pascal.

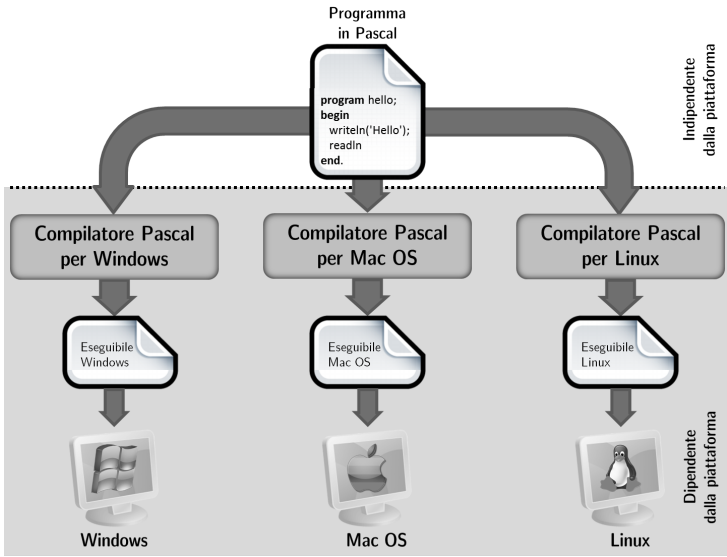


Figura 1.3: Un programma scritto nel linguaggio di una macchina astratta è portabile su più macchine reali.

Compilatori e interpreti

La macchina astratta $\mathcal{L}$ viene realizzata sulla macchina reale $\mathcal{M}$ utilizzando un opportuno strumento di **traduzione** dal linguaggio della macchina astratta al linguaggio della macchina reale. La traduzione può avvenire utilizzando due strumenti:

Compilatore è un programma che **traduce** un intero programma del linguaggio di $\mathcal{L}$ in un programma equivalente nel linguaggio macchina di $\mathcal{M}$ (vd. figura 1.2)

Interprete è un programma che simula direttamente la macchina astratta $\mathcal{M}$ leggendo un'istruzione alla volta del programma **P** scritto nel linguaggio di $\mathcal{L}$, effettuando le operazioni del linguaggio macchina di del linguaggio di $\mathcal{M}$ corrispondenti al suo significato e passando poi a considerare l'istruzione successiva.

L'introduzione della compilazione rende il codice sorgente (scritto nel linguaggio della macchina astratta $\mathcal{L}$) **portabile**, ovvero utilizzabile su tutte le macchine $\mathcal{M}_i$ per cui sia disponibile un compilatore del linguaggio $\mathcal{L}$ (vd. figura 1.3).

Lo schema standard di traduzione (vd. figura 1.4) prevede che il programma scritto nel linguaggio ad alto livello venga dapprima compilato, generando,

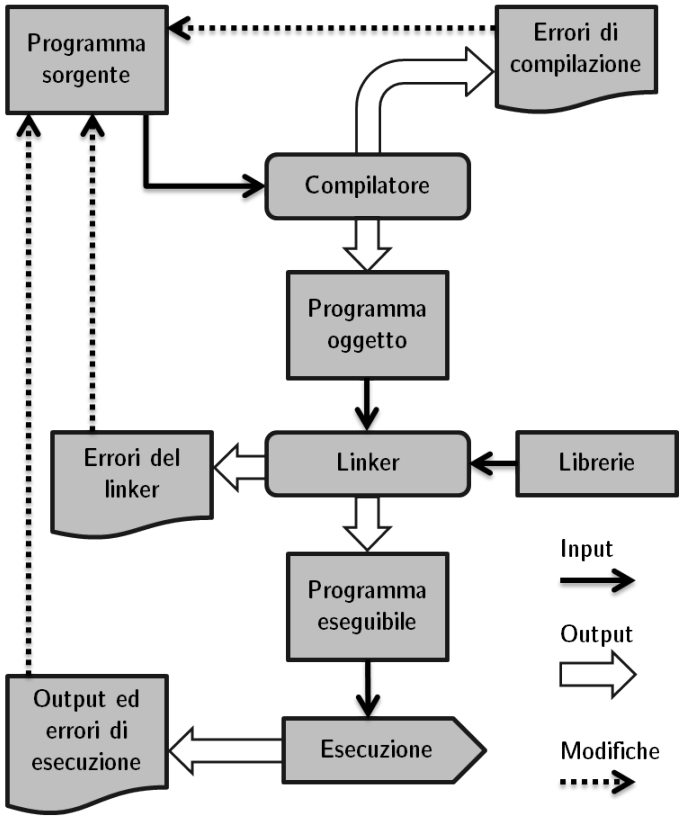


Figura 1.4: Dal programma in linguaggio ad alto livello al codice eseguibile.

eventualmente, gli errori ortografici e grammaticali che il compilatore è in grado di identificare. Il codice generato dalla compilazione (il codice **oggetto**) è la traduzione in linguaggio macchina solo del programma scritto dall'utente. A questo punto è necessario collegare (talvolta si dice *linkare*, dall'inglese *link*) il codice oggetto con il codice contenuto nelle librerie, ovvero con porzioni di codice scritto in linguaggio macchina che compiono operazioni standard quali, ad esempio, operazioni matematiche. Se il *linking* va a buon fine il codice è pronto per essere eseguito e testato sulla macchina di riferimento.

La *Java Virtual Machine* (JVM)

Un caso particolare di processo di traduzione è quello adottato dal linguaggio Java. Il codice sorgente, in questo caso, viene tradotto in un formato particolare, che non è strettamente legato ad una macchina reale, ma ad una macchina astratta standard $\mathcal{J}$. Il codice così generato, detto *bytecode*, viene poi interpretato,

istruzione per istruzione, nella macchina reale in cui deve essere eseguito. È a tutti gli effetti un processo di traduzione in due stadi (vd. figura 1.5).

Il vantaggio di quest'approccio è dato dalla portabilità completa del bytecode che può essere utilizzato su tutte le piattaforme che dispongono di un interprete Java che, normalmente, è un programma molto piccolo di dimensioni. Questo a patto che il codice venga pre-compilato utilizzando un compilatore da Java a bytecode che genera già un codice ben ottimizzato. Il bytecode così generato può anche essere facilmente trasmesso da un computer che lo produce (che effettua la compilazione) a un altro computer che lo utilizza (e ne effettua l'interpretazione). Questa caratteristica ha determinato la grande diffusione di Java come linguaggio per la programmazione di applicazioni web (*applet*).

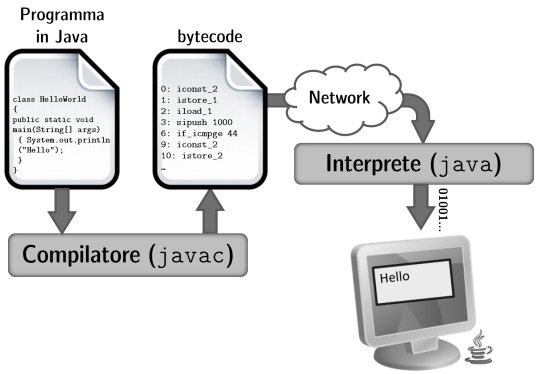


Figura 1.5: La traduzione in Java: la scelta di inserire il bytecode come codice intermedio consente di trasferire il file contenente il bytecode anche sul network e poi eseguire l'applicazione su qualunque macchina che ha a disposizione un interprete Java.

Capitolo 2

La programmazione e i linguaggi

2.1 La programmazione strutturata

2.1.1 Le strutture di controllo fondamentali

Per programmazione strutturata si intende una metodologia formalizzata e introdotta agli inizi degli anni settanta per dare delle linee guida ai programmatori al fine di seguire delle tecniche di sviluppo standardizzate. In questo contesto, l'esecutore è guidato alla sequenza di esecuzione opportuna, tra tutte quelle possibili, mediante tre **strutture di controllo** fondamentali:

Sequenza è il costrutto che prevede di eseguire le istruzioni secondo l'ordine lessicale in cui sono scritte all'interno del programma;

Selezione è il costrutto che guida la scelta dell'esecuzione di un blocco di istruzioni tra due possibili in base al valore di una condizione logica che viene valutata al momento della scelta;

Iterazione è il costrutto che permette di ripetere l'esecuzione di una o più istruzioni in base al valore di una condizione logica che viene ciclicamente controllata per decidere se proseguire l'iterazione o interromperla.

Impiegare in maniera accurata queste strutture migliora radicalmente la leggibilità dei programmi dato che ogni struttura di controllo ha un solo **punto di ingresso** e un solo **punto d'uscita** e il flusso di esecuzione è in qualche modo evidente dalla struttura del codice.

La giustificazione teorica che ha portato ad abbandonare progressivamente la programmazione in Assembler per passare alla programmazione strutturata, è il teorema di Böhm-Jacopini:

Teorema di completezza di Böhm-Jacopini (1966)

Tutti i programmi esprimibili tramite istruzioni di salto (***jump*** o ***goto***) o diagrammi di flusso (*flow-chart*) possono essere riscritti utilizzando esclusivamente le tre strutture di controllo fondamentali sequenza, selezione e iterazione.

Dopo la dimostrazione di questo teorema era chiaro che con linguaggi che fossero più legati a strumenti di ragionamento logico che all'architettura della macchina, sarebbe stato più semplice scrivere codice chiaro e modificabile.

Sequenza

Le istruzioni sono eseguite nello stesso ordine in cui compaiono nella lettura ordinata del programma, cioè secondo la **sequenza** in cui sono scritte.

Somma di due numeri

leggi i numeri a, b
calcola $a + b$
scrivi il risultato

Non è necessario definire neanche una vera e propria sintassi di questo costrutto, visto che è la modalità naturale in cui si eseguono le operazioni: leggendo le istruzioni una dopo l'altra, interprentandole ed eseguendole.

Selezione

Il costrutto di selezione è la definizione formale di una modalità naturale di affrontare la risoluzione dei problemi: se una certa condizione è vera si segue una certa strada per risolvere il problema, altrimenti, se è falsa, se ne segue un'altra. In termini sintattici e linguistici questo modello si sintetizza in maniera molto semplice.

Selezione

```
SE CONDIZIONE
    ALLORA
        BLOCCO1
    ALTRIMENTI
        BLOCCO2
FINESE
```

La spiegazione semantica della sintassi del costrutto è la seguente: viene valutata CONDIZIONE, se è **vera** vengono eseguite le istruzioni del BLOCCO1,

altrimenti, se è **falsa** vengono eseguite quelle del BLOCCO2. Al termine dell'esecuzione del costrutto si procede con l'istruzione immediatamente seguente la fine del costrutto di selezione (FINESE).

Una versione particolare del costrutto di selezione è quella che prevede di non fare niente quando la condizione di controllo è falsa.

Selezione (senza ALTRIMENTI)

```
SE CONDIZIONE
    ALLORA
        BLOCCO1
FINESE
```

La spiegazione semantica è simile a quella del costrutto base, ed è la seguente: viene valutata CONDIZIONE e se è **vera** vengono eseguite le istruzioni del BLOCCO1, altrimenti si procede con l'istruzione immediatamente seguente la fine del costrutto di selezione (FINESE).

Esempi di utilizzo del costrutto Si può usare il costrutto di selezione per verificare la condizione che il divisore sia diverso da zero, quando si effettua una divisione, per evitare di avere un errore aritmetico nello svolgimento dell'operazione.

Divisione tra due numeri (solo se il divisore è diverso da zero)

```
leggi il dividendo e il divisore
SE il divisore è diverso da zero
    ALLORA
        calcola dividendo/divisore
        scrivi il risultato
    ALTRIMENTI
        scrivi "Errore:  divisione per zero"
FINESE
```

I costrutti possono essere anche annidati l'uno dentro l'altro, ovvero un blocco interno di un costrutto di selezione può essere, a sua volta, un altro costrutto di selezione. Non si fa altro che simulare il metodo di ragionamento che porta a costruire un albero di decisioni in cui si hanno varie condizioni da verificare in cascata. In questo esempio come prima decisione si controlla se, nella soluzione di un'equazione di secondo grado, il discriminante sia uguale o diverso da zero. Se è uguale a zero, allora si può verificare se il numero di soluzioni reali sia uno o due.

Calcolo delle radici di $ax^2 + bx + c = 0$

leggi i valori di a, b, c e calcola il discriminante $\Delta = b^2 - 4ac$
SE Δ è minore di zero
  ALLORA
    scrivi "**Nessuna soluzione reale**"
  ALTRIMENTI
    SE Δ è uguale a zero
      ALLORA
        calcola $\frac{-b}{2a}$
        scrivi "**Due soluzioni coincidenti:**", il risultato
      ALTRIMENTI
        calcola $\frac{-b-\sqrt{b^2-4ac}}{2a}$ e $\frac{-b+\sqrt{b^2-4ac}}{2a}$
        scrivi "**Due soluzioni distinte:**", i due risultati
    FINESE
FINESE

Iterazione con controllo finale

Il controllo che prevede di effettuare un'iterazione, una ripetizione, di un insieme di istruzioni è il più complesso e variegato dei costrutti fondamentali della programmazione strutturata. In questo caso la verifica della condizione può avvenire sia all'inizio che alla fine dell'esecuzione del corpo del costrutto.

Iterazione con controllo finale

ESEGUI
    BLOCCO
FINQUANDO CONDIZIONE

La spiegazione semantica è la seguente: viene **sicuramente** eseguito BLOCCO, dopodiché viene valutata CONDIZIONE, se è **vera** si ritorna al passo precedente, altrimenti, se è **falsa** l'esecuzione riprende dalla prima istruzione che segue il costrutto iterativo.

Abbiamo quindi la sicurezza che BLOCCO sia eseguito **almeno una volta** e che l'esecuzione termini quando CONDIZIONE diventa falsa.

Esempi di utilizzo del costrutto Se vogliamo calcolare la somma dei primi 100 numeri interi dobbiamo effettuare sempre la stessa operazione, sommare il numero corrente alla somma parziale, per un numero di volte pari al numero di interi da sommare e, ad ogni passo, ricordarci che aumentiamo di uno il numero da prendere in considerazione.

Somma dei primi 100 numeri interi: calcolo iterativo, senza utilizzare la formula di Gauss ($\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$)

```
poni il valore della somma a zero
prendi come numero corrente il numero 1
ESEGUI
    aggiungi alla somma il numero corrente
    passa al numero successivo
FINQUANDO il numero corrente non supera 100
scrivi la somma
```

Iterazione con controllo iniziale

Nella forma alternativa, che prevede il controllo iniziale, la computazione parte con la valutazione di CONDIZIONE:

Iterazione con controllo iniziale

```
FINQUANDO CONDIZIONE ESEGUI
    BLOCCO
RIPETI
```

La spiegazione semantica è diversa ed è la seguente: viene valutata CONDIZIONE, se è **vera** viene eseguito BLOCCO e quindi si torna al passo precedente, se invece è **falsa** l'esecuzione riprende dalla prima istruzione che segue il costrutto iterativo.

In questo caso il BLOCCO può essere eseguito anche **zero** volte mentre continua ad essere vero che l'esecuzione del costrutto termina quando la CONDIZIONE diventa falsa.

Tra i due schemi del costrutto iterativo esiste un'equivalenza semantica che può essere dimostrata introducendo una combinazione di costrutto con controllo finale e selezione.

Il comportamento dello schema (`FINQUANDO...RIPETI`) può essere infatti simulato combinando lo schema (`ESEGUI...FINQUANDO...`) con un costrutto di selezione:

```
SE CONDIZIONE
   ALLORA
      ESEGUI
         BLOCCO
      FINQUANDO CONDIZIONE
FINESE
```

2.1.2 Variabili e assegnamento

Nel contesto dei linguaggi di programmazione con il nome **variabile** si definisce un generico *contenitore* astratto il cui compito è quello di contenere dei valori. Il metodo base per modificare il valore di una variabile è utilizzare un'istruzione di assegnamento.

Assegnamento

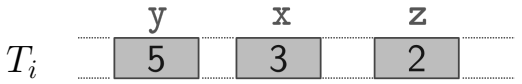
variabile ← espressione

La sua semantica è:

1. Viene calcolato il valore dell'espressione scritta a destra del simbolo ←;
2. Il risultato ottenuto è assegnato alla variabile (quindi posto nel contenitore) il cui nome è scritto a sinistra del simbolo ←, sostituendo l'eventuale valore in essa precedentemente contenuto.

È importante notare come molti linguaggi, tra cui anche C, utilizzino purtroppo per l'assegnamento il simbolo =, usato in aritmetica per indicare l'uguaglianza. Questo, come vedremo, può essere fonte di errori anche molto difficili da individuare.

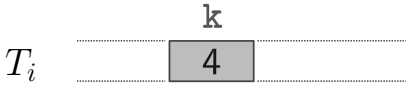
Assegnamento: caso base



`x ← y + z`

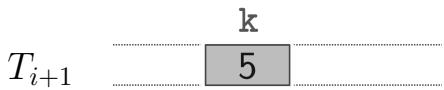
Per prima cosa si recuperano i valori presenti in `y` (5) e `z` (2) e si **valuta l'espressione** `y+z` (ovvero, si calcola la somma). Poi il risultato ottenuto viene scritto nel **contenitore** denominato `x`, sostituendo il valore che è già presente.

Assegnamento: riuso del valore



```
k ← k + 1
```

Anche in questo si recupera il valore di `k` (4) e poi si valuta l'espressione `k+1`, sommando `1` al valore recuperato. Il risultato ottenuto viene scritto nel **contenitore** denominato `k` sostituendo il valore presente. In questa istruzione il simbolo `k` viene utilizzato sia per indicare il suo contenuto che per indicare il contenitore. Questo sarà sempre vero con le variabili che si utilizzano all'interno di un linguaggio di programmazione.



Tipo di una variabile

Il **tipo** di una variabile specifica la classe di valori che questa può assumere e l'insieme delle operazioni che su di essa possono essere effettuate.

Ad esempio una variabile `x` di tipo **intero** può assumere come valori solo **numeri interi** e su di essa possono essere effettuate soltanto **le operazioni** consentite per i numeri interi, una radice quadrata non rientrerà tra queste.

Sebbene tutte le variabili siano rappresentate nella memoria come sequenze di bit, tali sequenze possono essere interpretate diversamente in base ai tipi. La nozione di tipo fornisce un'astrazione rispetto alla rappresentazione effettiva dei dati. Il programmatore può utilizzare variabili di tipi differenti, senza necessità di conoscerne l'effettiva rappresentazione.

Variabile come astrazione

Il concetto di **variabile** è un'**astrazione** della locazione di memoria. Se in un computer, fisicamente, si dovesse accedere alla locazione di memoria se ne dovrebbe conoscere l'indirizzo esatto. Passando alla sua astrazione, ci si può permettere di utilizzare un nome simbolico, la cui corrispondenza con una locazione fisica di memoria sarà determinata da qualche sistema accessorio di cui il programmatore non si deve interessare.

Allo stesso modo l'assegnamento di un valore a una variabile è un'astrazione dell'operazione **`STORE`**, tipica dei linguaggi Assembler che, come abbiamo visto, aveva come uno dei suoi parametri l'indirizzo fisico della locazione di memoria utilizzata.

Dichiarazione delle variabili

La maggior parte dei linguaggi di programmazione richiedono di **dichiarare** le variabili utilizzate nel programma indicandone il tipo. La modalità di dichiara-

zione può variare, ad esempio Pascal richiede che le variabili siano dichiarate tutte all'inizio del programma.

Questa scelta ha una serie di vantaggi: accresce la leggibilità dei programmi, diminuisce la possibilità di errori e facilita la realizzazione di compilatori efficienti. L'unico svantaggio evidente è invece la diminuzione di flessibilità dovuta alla pianificazione dell'utilizzo.

2.1.3 Esempi di utilizzo delle variabili per la soluzione di semplici problemi

Calcolo delle radici di $ax^2 + bx + c = 0$

La prima stesura dell'algoritmo può essere effettuata lasciando in linguaggio molto "naturale" la descrizione di alcune attività, inserendo, comunque, le descrizioni all'interno dei costrutti come abbiamo visto nell'esempio in 2.1.1. Questa descrizione è già molto simile alla definizione dell'algoritmo che potrà essere scritta nel linguaggio di programmazione scelto, perlomeno per quanto riguarda il tipo di costrutti da utilizzare.

In un secondo passaggio si può ipotizzare quali e quante variabili possano essere utilizzate per schematizzare i valori da utilizzare per i calcoli e riscrivere l'algoritmo di conseguenza:

```
variabili a, b, c, delta, x1, x2: numeri reali

leggi a, b, c
delta ← b^2 - 4·a·c
SE delta < 0
  ALLORA
    scrivi "Nessuna soluzione reale"
  ALTRIMENTI
    SE delta = 0
      ALLORA
        x1 ← -b / (2·a)
        scrivi "Due soluzioni coincidenti:  ", x1
      ALTRIMENTI
        x1 ← (-b - √delta) / (2·a)
        x2 ← (-b + √delta) / (2·a)
        scrivi "Due soluzioni:  ", x1, x2
    FINESE
FINESE
```

Somma di n numeri interi consecutivi

Il processo di risoluzione di un problema non necessariamente deve essere monolitico e avere un unico stadio di maturazione. Supponiamo di voler sviluppare un algoritmo che calcoli la somma di un certa quantità di numeri interi. Il primo passo che possiamo fare può essere di risolvere un problema che abbia un parametro in meno, ovvero il calcolo della somma di un numero predefinito di interi, in questo caso 100.

Somma dei numeri da 1 a 100

```
poni il valore della somma a zero
inizia a considerare il numero 1
ESEGUI
   aggiungi alla somma il numero che stai considerando
   considera il numero successivo
FINQUANDO il numero che stai considerando non supera 100
scrivi la somma
```

A questo punto possiamo introdurre l'insieme di variabili necessario a schematizzare in maniera ulteriore il processo risolutivo. Ancora una volta, questo passaggio non deve cambiare la strategia risolutiva che è stata oramai definita al momento della stesura dell'algoritmo. Quello che dobbiamo fare è la traduzione dell'algoritmo da una lingua meno formale (la nostra lingua madre, in questo caso l'italiano) a una lingua più formale, che non è ancora però un vero e proprio linguaggio di programmazione. Non esiste infatti una macchina astratta che è in grado di eseguire questo programma.

```
variabili somma, cont: numeri interi

somma ← 0
cont ← 1
ESEGUI
  somma ← somma + cont
  cont ← cont + 1
  FINQUANDO cont ≤ 100
scrivi somma
```

Possiamo adesso facilmente generalizzare al calcolo della somma primi n numeri interi.

```
variabili somma, cont, n: numeri interi

leggi n
somma ← 0
cont ← 1
```

```
ESEGUI
  somma ← somma + cont
  cont ← cont + 1
  FINQUANDO cont ≤ n
scrivi somma
```

A questo punto possiamo fare degli aggiustamenti alla soluzione per tener conto di casi particolari che, in prima approssimazione, possiamo aver tralasciato. Dobbiamo, in altre parole, fare attenzione alla **correttezza** del risultato. Con la soluzione proposta, se $n \leq 0$ il risultato non è corretto: dovrebbe essere 0 ed invece si ottiene 1, quindi conviene cambiare modello di struttura di iterazione passando dal primo al secondo modello.

```
variabili somma, cont, n: numeri interi

leggi n
somma ← 0
cont ← 1
FINQUANDO cont ≤ n ESEGUI
  somma ← somma + cont
  cont ← cont + 1
RIPETI
scrivi somma
```

Adesso possiamo ulteriormente estendere la soluzione proposta al calcolo della somma di n interi consecutivi a partire da k.

```
variabili somma, cont, n, k: numeri interi

leggi n, k
somma ← 0
cont ← 1
FINQUANDO cont ≤ n ESEGUI
  somma ← somma + k
  k ← k + 1
  cont ← cont + 1
RIPETI
scrivi somma
```

Numeri pari o dispari

Iniziamo con la stesura del semplice algoritmo che, dato un numero n, indica se è pari o dispari.

Determina se un numero è pari o dispari

leggi il numero n
calcola il resto della divisione di n per 2
SE il resto è uguale a zero
  ALLORA
    scrivi "**pari**"
  ALTRIMENTI
    scrivi "**dispari**"
FINESE

Traduciamo in un linguaggio più simile a un linguaggio di programmazione e aggiungiamo le variabili.

```
variabili n, resto: numeri interi

leggi n
resto ← n MOD 2
SE resto = 0
  ALLORA
    scrivi "pari"
  ALTRIMENTI
    scrivi "dispari"
FINESE
```

Possiamo adeso **compattare** le operazioni per rendere il codice più sintetico e in qualche modo più efficiente, risparmiando l'uso della variabile (`resto`).

```
variabili n: numeri interi

leggi n
SE n MOD 2 = 0
  ALLORA
    scrivi "pari"
  ALTRIMENTI
    scrivi "dispari"
FINESE
```

Il costo che paghiamo è quello di rendere il codice un po' **meno leggibile**. Questo ***trade-off*** tra efficienza e leggibilità è, d'altronde, un elemento costante nel mondo della programmazione.

Impariamo adesso dalla soluzione del problema semplice per risolverne un altro simile ma più complesso. Passiamo al problema di leggere una sequenza di numeri interi e contare i pari e i dispari sinché non si incontra uno zero.

Conta i pari e dispari in una sequenza di numeri interi

```
azzera i due contatori dei numeri pari e dispari
leggi un numero
FINQUANDO il numero è diverso da zero ESEGUI
  SE il numero è pari
    ALLORA
      incrementa il contatore dei numeri pari
    ALTRIMENTI
      incrementa il contatore dei numeri dispari
  FINESE
  leggi un numero
RIPETI
scrivi i valori dei contatori
```

Dovrebbe adesso essere semplice l'operazione di traduzione con l'introduzione delle variabili adeguate.

```
variabili numero, npari, ndispari: numeri interi

npari ← 0
ndispari ← 0
leggi numero
FINQUANDO numero ≠ 0 ESEGUI
  SE numero MOD 2 = 0
    ALLORA
      npari ← npari + 1
    ALTRIMENTI
      ndispari ← ndispari + 1
  FINESE
  leggi numero (* il prossimo della sequenza *)
RIPETI
scrivi npari, ndispari
```

2.2 Sintassi e semantica

I due livelli di interpretazione del linguaggio, sintassi e semantica, possono essere entrambi descritti in maniera formale per quanto riguarda i linguaggi di programmazione. Mentre la descrizione della sintassi è un'operazione semplice e basilare, la formalizzazione della semantica è molto più complessa e richiede un livello di approfondimento maggiore. Qui tratteremo solo le modalità di descrizione della sintassi.

2.2.1 Sintassi di un linguaggio

La sintassi di un linguaggio è l'insieme delle sue regole grammaticali. Se per i linguaggi cosiddetti "naturali", quali, ad esempio, l'italiano, questo insieme di regole può essere flessibile, mutevole e in continua evoluzione, per i linguaggi di programmazione è un insieme definito a priori una volta per tutte dai progettisti del linguaggio, quindi immutabile e da rispettarsi in maniera rigida. La grammatica specifica **come si scrivono** le frasi del linguaggio.

Esistono varie notazioni per descrivere la sintassi dei linguaggi, le due più utilizzate all'interno dei linguaggi di programmazione sono la forma di Backus e Naur (Backus-Naur Form, BNF) e i grafi sintattici.

BNF

Definiamo due categorie di simboli che possono essere utilizzati all'interno delle definizioni BNF: i **simboli terminali** T e le **categorie sintattiche** N. Nel paragone con i linguaggi naturali, T rappresenta il dizionario del linguaggio, mentre N le categorie di elementi della frase (es. verbi, articoli, ecc.).

Vediamo un semplice esempio che utilizza parole e categorie della lingua italiana.

T={`il`, `lo`, `la`, `cane`, `tenda`, `gatto`,`mangia`, `graffia`, **E**}

N={Frase, Soggetto, Verbo, Complemento,Articolo, Nome}

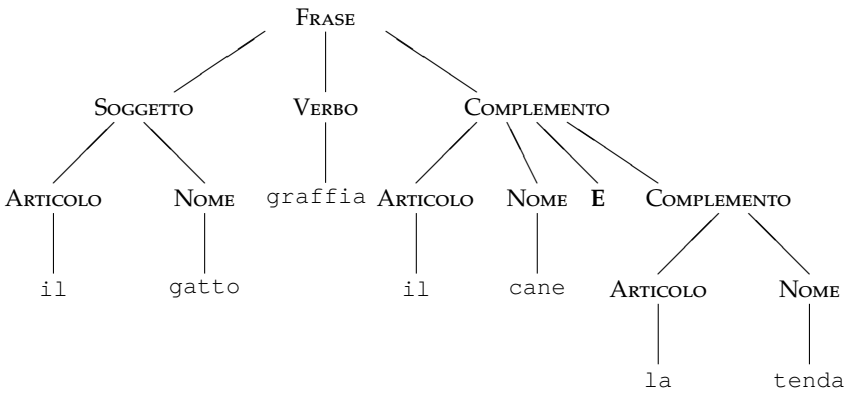
Le regole di produzione (P), ovvero di trasformazione di categorie in concatenazioni di altre categorie e terminali, scritte con la convenzione BNF sono le seguenti:

1. Frase ::= Soggetto Verbo Complemento
2. Soggetto ::= Articolo Nome
3. Articolo ::= `il` | `la` | `lo`
4. Nome ::= `cane` | `tenda` | `gatto`
5. Verbo ::= `mangia` | `graffia`
6. Complemento ::= Articolo Nome | Articolo Nome **E** Complemento

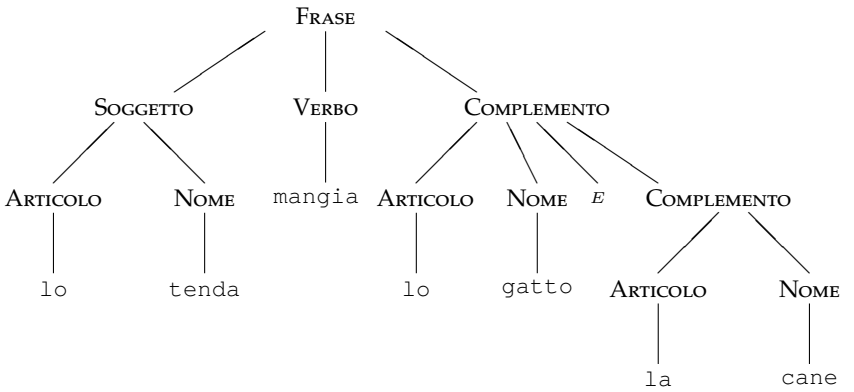
Il simbolo '::=' indica la possibilità di sostituire al simbolo sulla sinistra una delle alternative elencate sulla destra, divise da |. Ad esempio, la regola n. 2 ci dice che un Soggetto è costituito da un Articolo seguito da un Nome.

All'interno delle categorie sintattiche ne esiste una speciale, che genera ogni elemento del linguaggio. In questo esempio è $frase$.

Partendo dal simbolo iniziale (Frase) si possono operare una serie di sostituzioni sino ad ottenere una concatenazione formata solo da simboli terminali, una frase sintatticamente corretta del linguaggio:



Ottenere una frase sintatticamente corretta non garantisce niente sulla generazione di una frase che abbia un *significato* nel linguaggio, ovvero che abbia un senso dal punto di vista semantico. Quella che segue è una frase corretta nella sintassi, ma senza senso per la semantica della lingua italiana:



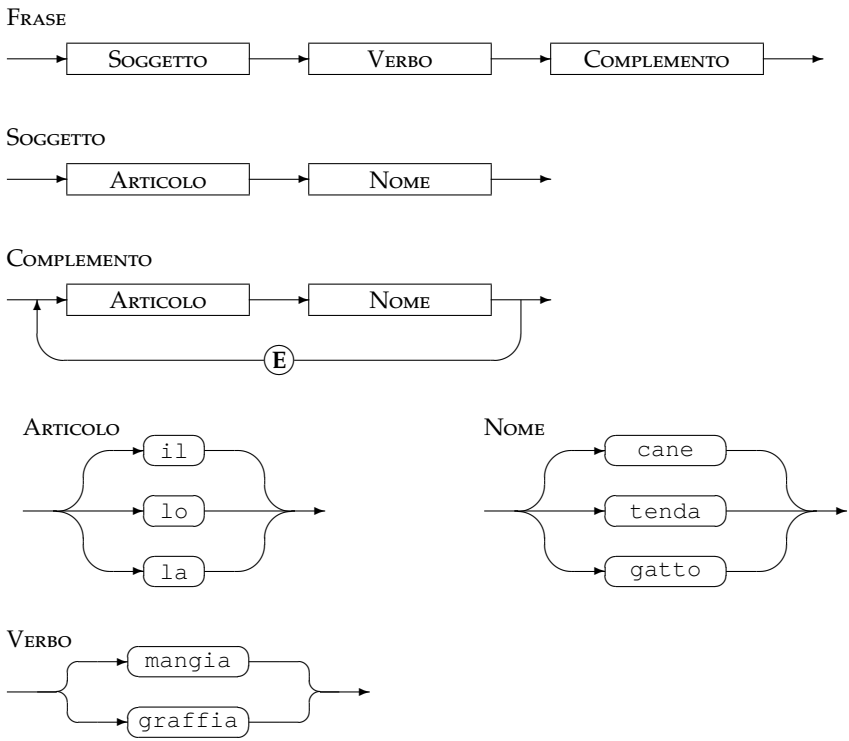
Questo è vero anche nel campo dei linguaggi di programmazione: la correttezza sintattica, garantita dal controllo effettuato dal compilatore, non ci dà alcuna garanzia sull'aver scritto un programma che risolve il problema che ci si era riproposti di risolvere.

Grafi sintattici

Il formalismo dei grafi sintattici può essere utilizzato in alternativa al modello BNF per rappresentare le regole di produzione. Si specifica una *carta sintattica* per ciascun simbolo non terminale della grammatica.

In un grafo sintattico i **rettangoli** indicano simboli non terminali (che andranno espansi con i grafi sintattici corrispondenti) e gli **ovali** indicano simboli terminali, che, quindi. non devono essere espansi ulteriormente. Ogni **biforcazione** indica un'alternativa (è il corrispondente del simbolo | in BNF).

Il grafo sintattico che descrive il medesimo linguaggio descritto in precedenza in BNF è il seguente.



2.2.2 Il dizionario del linguaggio C

Passando ad un linguaggio di programmazione concreto, come il linguaggio C, si può iniziare a definire il linguaggio partendo dall'**Alfabeto**utilizzato per il linguaggio. L'alfabeto del C si chiama *Unicode* ed è un insieme di caratteri ognuno dei quali è rappresentato con 16 bit.

Il dizionario del linguaggio è costituito da due tipi di parole: le **Parole riservate (o parole chiave)** e i simboli definiti dall'utente.

Parole chiave

Sono quelle parole che nel linguaggio hanno un significato predeterminato: non possono essere utilizzate diversamente da come sono state definite dai progettisti del linguaggio e, quindi, non possono essere ridefinite.

Le seguenti sono tutte le parole del dizionario delle parole riservate del C[1]:

[1] in **grassetto** quelle che sono trattate in queste dispense.

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

Identificatori

Sono nomi impiegati all'interno del programma per indicare variabili, funzioni, nuovi tipi, ecc. Sono scelti dall'utente seguendo le regole grammaticali del linguaggio, senza sovrapporsi con le parole riservate. In C un identificatore è costituito da una sequenza di lettere e cifre che inizia con una lettera.

Separatori

Sono caratteri che permettono di separare o raggruppare parti di codice.

```
{    }    [    ]    (    )    .    ,    :    ;
```

Operatori

Sono simboli o sequenze di simboli che denotano operazioni effettuate tramite costrutti del linguaggio.

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

Letterali

Sono detti più comunemente costanti e sono sequenze di caratteri utilizzate all'interno dei programmi per rappresentare valori di tipi primitivi e stringhe.

Commenti

Ci sono due modalità di inserire commenti all'interno di codice C:

commenti che si estendono su più righe di testo iniziano con i caratteri **/*** e terminano con i caratteri ***/**; il compilatore ignora tutto il testo compreso tra questi caratteri all'interno di questi due delimitatori.

commenti a fine riga si aprono con la coppia di caratteri **//** e si chiudono alla fine della riga; il compilatore ignora tutto il testo che inizia dai caratteri **//** fino alla fine della riga

2.3 La storia del linguaggio C

2.3.1 UNIX e C

Gli inizi

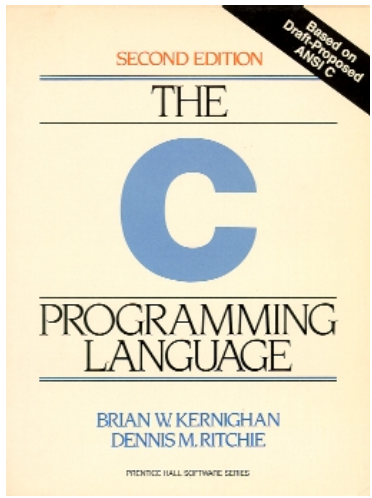
Il linguaggio C nasce con un'esigenza precisa: aiutare nella scrittura del sistema operativo UNIX, Per dare un inquadramento cronologico, si deve ricordare che siamo alla fine degli anni '60 e la programmazione di basso livello era fatta ancora praticamente solo in Assembler. La grande intuizione dei progettisti di UNIX, Thompson Ritchie e altri, fu quella di scrivere il sistema operativo utilizzando un linguaggio ad alto livello. Questa scelta garantì la possibilità di avere il sistema operativo disponibile su tutte le macchine scrivendo in Assembler solo il compilatore per il linguaggio ad alto livello.

Il problema di secondo livello era rappresentato dalla mancanza di un linguaggio adatto a questo scopo. Il sottoprodotto della progettazione di UNIX fu, dunque, la definizione di un linguaggio di programmazione che garantisse, allo stesso tempo, la portabilità di un linguaggio ad altro livello e la flessibilità ed efficienza di un linguaggio assemblativo: questo linguaggio fu chiamato C.

Tappe dello sviluppo del linguaggio

1973 Prima versione del C elaborata da Dennis Ritchie ai Bell Labs per programmare UNIX;

1978 Esce il Kerningham&Ritchie, libro che descrive per la prima volta il linguaggio e definisce il primo standard *de facto* per il linguaggio detto **K&R C**



1989 Viene approvato ANSI C, primo standard vero e proprio per il linguaggio che diviene uno standard ISO dall'anno successivo noto come **C89**;

1999 Viene rivista la versione ufficiale dello standard, detta **C99**;

2011 È l'anno dell'ultima e corrente versione, che viene chiamato **C11**.

Poiché lo sviluppo di codice prima del C99 e del C11 è stato enorme (si parla di miliardi di linee di codice) ancora oggi C89 è lo stile C forse più diffuso.

2.3.2 Linguaggi derivati dal C

La grande popolarità del linguaggio C ha fatto sì che molti linguaggi progettati in tempi successivi si siano ispirati a C anche solo nell'ortografia o, più sostanzialmente, siano delle evoluzioni dirette del linguaggio. Una lista non esaustiva comprende:

C++ è sostanzialmente un'evoluzione ed un'integrazione del C verso il paradigma *object-oriented*

Java ancora più aderente ai dettami OO e fortemente orientato ad applicazioni distribuite

C# progetto di evoluzione del C verso il paradigma object-oriented, in concorrenza con C++ elaborato in ambiente Microsoft, legato anche a F# (linguaggio funzionale)

Perl scripting language molto popolare per applicazioni legate in particolare alla manipolazione di stringhe e di testo

È importante notare che C è, per certi versi, più rudimentale di tutti questi linguaggi, ma che, comunque,il modo in cui si utilizza un linguaggio è ben più importante del linguaggio in sé. Un problema non si risolve con il linguaggio, ma progettando l'algoritmo corretto che lo risolve, poi la scelta del linguaggio adeguato semplifica la realizzazione del programma accelerando lo sviluppo.

2.3.3 Pregi e difetti

Caratteristiche

1. C è un linguaggio (troppo!) di basso livello, consente di accedere a caratteristiche del computer che hanno un senso se si programma un sistema operativo altrimenti devono rimanere nascoste, è rischioso utilizzarle

2. La sintassi del C è molto semplice, con un numero limitato di istruzioni, fa ampio uso di funzioni di libreria

3. I controlli che vengono effettuati (sia dal compilatore che dal linker) quando si scrivono programmi in C sono minori di quanto desiderabile e non esiste un meccanismo di eccezioni ⇒ maggiore libertà ma anche maggior rischio di errori

Pregi

Efficienza Diretta derivazione dagli scopi per cui nasce

Portabilità Caratteristica ampliata con il tempo, dalla disponibilità di compilatori (piccoli e facili da scrivere) per tutte le piattaforme (mobile incluso) e dalla standardizzazione del linguaggio (no dialetti incompatibili)

Flessibilità C è (stato) utilizzato per programmare le applicazioni più disparate, sia per la sua larga diffusione tra i programmatori sia per la possibilità di compiere operazioni impensabili con altri linguaggi

UNIX/Linux C nasce con UNIX e di UNIX rimane una componente fondamentale, avere a che fare con Linux senza conoscere C è quasi impossibile

Difetti

Error proneness La grande flessibilità è anche un grande difetto, mancano controlli di consistenza che altri linguaggi hanno in maniera molto efficiente, si possono introdurre errori nel codice praticamente impossibili da individuare

Obfuscation Piccolo insieme di comandi e operatori che però, mescolati in maniera accorta, possono costruire programmi pressoché completamente incomprensibili, sia per scelta che per scarsa volontà di documentazione

Manutenzione C manca di tutti quegli strumenti che consentono una suddivisione razionale dello sviluppo del codice, anche tra sviluppatori diversi, per questo può essere anche molto difficile effettuare la **manutenzione** del codice

Un codice totalmente incomprensibile

Il seguente programma è effettivamente compilabile ed eseguibile[2], seguendo dei particolari accorgimenti per dare informazioni al compilatore su come trattare i costrutti utilizzati. È un esempio pratico di come si riesca a scrivere codice praticamente incomprensibile.

[2] È un'implementazione molto semplice di un algoritmo di ray-tracing che consente di realizzare delle immagini foto-realistiche di una scena digitale, dopo aver dato la definizione degli oggetti presenti e delle sorgenti di illuminazione.

```
X=1024; Y=768; A=3;

J=0;K=-10;L=-7;M=1296;N=36;O=255;P=9;_=1<<15;E;S;C;D;F(b){E="1""111886:6:??AAF"
"FHHMMOO55557799@@>>>BBBGGIIKK"[b]-64;C="C@=::C@@==@=:C@=:C@=:C5""31/513/5131/"
"31/531/53"[b ]-64;S=b<22?9:0;D=2;}I(x,Y,X){Y?(X^=Y,X*X>x?(X^=Y):0,  I (x,Y/2,X
)):(E=X);       }H(x){I(x,     _,0);}p;q(         c,x,y,z,k,l,m,a,           b){F(c
);x-=E*M      ;y-=S*M            ;z-=C*M          ;b=x*        x/M+          y*y/M+z
*z/M-D*D     *M;a=-x              *k/M      -y*l/M-z         *m/M;     p=((b=a*a/M-
b)>=0?(I     (b*M,_       ,0),b     =E,       a+(a>b       ?-b:b)):      -1.0);}Z;W;o
(c,x,y,      z,k,l,     m,a){Z=!     c?       -1:Z;c      <44?(q(c,x          ,y,z,k,
l,m,0,0      ),(p>       0&&c!=      a&&          (p<W          ||Z<0)           )?(W=
p,Z=c):      0,o(c+          1,     x,y,z,          k,l,           m,a)):0      ;}Q;T;
U;u;v;w     ;n(e,f,g,             h,i,j,d,a,     b,V){o(0       ,e,f,g,h,i,j,a);d>0
&&Z>=0? (e+=h*W/M,f+=i*W/M,g+=j*W/M,F(Z),u=e-E*M,v=f-S*M,w=g-C*M,b=(-2*u-2*v+w)
/3,H(u*u+v*v+w*w),b/=D,b*=b,b*=200,b/=(M*M),V=Z,E!=0?(u=-u*M/E,v=-v*M/E,w=-w*M/
E):0,E=(h*u+i*v+j*w)/M,h-=u*E/(M/2),i-=v*E/(M/2),j-=w*E/(M/2),n(e,f,g,h,i,j,d-1
,Z,0,0),Q/=2,T/=2,        U/=2,V=V<22?7:  (V<30?1:(V<38?2:(V<44?4:(V==44?6:3))))
,Q+=V&1?b:0,T                 +=V&2?b         :0,U+=V     &4?b:0)      :(d==P?(g+=2
,j=g>0?g/8:g/      20):0,j     >0?(U=      j     *j/M,Q       =255-     250*U/M,T=255
-150*U/M,U=255     -100     *U/M):(U     =j*j      /M,U<M            /5?(Q=255-210*U
/M,T=255-435*U             /M,U=255     -720*       U/M):(U         -=M/5,Q=213-110*U
/M,T=168-113*U     /        M,U=111                  -85*U/M)        ),d!=P?(Q/=2,T/=2
,U/=2):0);Q=Q<     0?0:       Q>O?      O:           Q;T=T<0?     0:T>O?O:T;U=U<0?0:
U>O?O:U;}R;G;B     ;t(x,y      ,a,     b){n(M*J+M     *40*(A*x    +a)/X/A-M*20,M*K,M
*L-M*30*(A*y+b)/Y/A+M*15,0,M,0,P,  -1,0,0);R+=Q     ;G+=T;B    +=U;++a<A?t(x,y,a,
b):(++b<A?t(x,y,0,b):0);}r(x,y){R=G=B=0;t(x,y,0,0);x<X?(printf("%c%c%c",R/A/A,G
/A/A,B/A/A),r(x+1,y)):0;}s(y){r(0,--y?s(y),y:y);}main(){printf("P6\n%i %i\n255"
"\n",X,Y);s(Y);}
```

2.4 Anatomia di un programma C

2.4.1 Struttura generale

Un programma C, semplice o complesso che sia, ha sempre la stessa struttura di base

```
direttive di pre-processing

int main(void)
{                ← inizia il blocco di istruzioni
    istruzioni
}                ← finisce il blocco di istruzioni
```

I caratteri `{` e `}` sono la coppia di **delimitatori di blocco** e hanno la funzione che in molti altri linguaggi è assolta da parole ben più significative come `begin` e `end`, ovvero di dichiarare quando inizia e quando termina un blocco di istruzioni. Vedremo nel seguito quale sarà la funzione di identificare chiaramente e senza ambiguità i blocchi che costituiscono un programma e le sue parti.

2.4.2 Preprocessing

Per pre-processing si intende tutta l'attività di trattamento delle informazioni contenute nel programma che precede la vera e propria compilazione.

Direttive di preprocessing

Sono informazioni che vengono utilizzate da uno strumento di sostituzione di porzioni di testo in altre porzioni: il preprocessore. Compito del preprocessore (*pre-processor*) è quello di trasformare il testo del programma prima della compilazione.

Le due direttive più utilizzate, che sono presenti praticamente in ogni programma, sono:

- `#include`
- `#define`

Per regola sintattica le direttive si trovano su una sola riga di codice e non hanno nessun carattere delimitatore né di blocco né di istruzione.

La direttiva `#include` è utilizzata per includere all'interno del proprio programma informazioni relative alle funzioni contenute in parti della libreria standard

Inclusione mediante direttiva di pre-processing

```
#include <stdio.h>[a]
```

[a] Questo comando ci permette di dichiarare, e poi utilizzare senza doverle definire, tutte le funzioni che vengono utilizzate per l'I/O

La direttiva `#define` serve per dare un nome simbolico a valori che vengono utilizzati, di solito più volte, all'interno del programma.

Definizione di macro

```
#define PI 3.1415[a]
```

[a] Questo comando ci permette di utilizzare il nome simbolico `PI` all'interno del programma, anziché il valore numerico che gli è associato

2.4.3 Comandi e funzioni

Comandi

Le varie tipologie di comandi che possono essere utilizzate all'interno di un programma si possono classificare nelle categorie seguenti:

Dichiarazioni Si trovano, per buona norma, all'inizio del programma e definiscono gli oggetti (variabili e altro) con i loro identificatori che verranno utilizzati all'interno del codice; ogni oggetto ha un tipo associato;

Istruzioni di assegnamento Consentono di variare il valore di una variabile o, per meglio dire, di un *lvalue*, che vedremo nel seguito cosa significa;

Struttura di selezione Definisce quali istruzioni, tra le varie alternative (due o più) debbano essere scelte come proseguimento dell'esecuzione;

Strutture di iterazione Consentono di definire, in modo compatto, l'esecuzione della stessa operazione per un numero prefissato o deciso durante il corso dell'esecuzione di volte;

Chiamate di funzioni Passano il controllo dell'esecuzione ad un'altra porzione di codice (funzione, subroutine o sottoprogramma) che svolge un compito simile a quello del programma nel suo complesso, da un insieme di dati in input genera un output che restituisce.

Funzioni

Le funzioni sono il principale meccanismo per mezzo del quale un programma C può essere suddiviso in sezioni distinte, ognuna delle quali svolge un'attività specifica.

Per comodità di classificazione possiamo dividerle in due categorie, anche se la distinzione è puramente legata alla modalità con cui il programmatore ne dispone, non certo alla sostanza:

1. Funzioni che il programmatore scrive per comodità di progettazione
2. Funzioni disponibili nella libreria standard o in altre librerie utilizzate ad-hoc

Il primo tipo deriva dalla necessità, da parte dello sviluppatore dell'applicazione, di suddividere il proprio sistema in un insieme di elementi, ognuno dei quali risolva una parte del problema globale. Si tratta, quindi, di adeguarsi a un requisito di **modularità**.

Il secondo tipo comprende le funzioni che sono disponibili per tutti gli utenti; il singolo programmatore, per utilizzarle, ne deve conoscere la sintassi che è disponibile all'interno dei file *header* `.h`. La loro implementazione si trova all'interno delle librerie ed è il linker che le mette a disposizione del programma che le invoca, nel caso della libreria standard senza neanche si debba dare alcuna direttiva esplicita al riguardo.

2.4.4 Una digressione sul termine funzione

Funzioni e linguaggi funzionali

Nell'ambito dei linguaggi **funzionali** il termine **funzione** ha un'accezione abbastanza diversa rispetto a quella che ha nel linguaggio C. Programmare in un linguaggio funzionale consiste nel costruire definizioni e usare il calcolatore per valutare espressioni, l'obiettivo di un programmatore è quello di progettare

una funzione, normalmente espressa in termini matematici e che può fare uso di un certo numero di funzioni ausiliarie, per risolvere un problema dato.

In quest'ottica, potremmo dire che il calcolatore si utilizza come fosse una calcolatrice tascabile, quello che distingue un calcolatore da una normale calcolatrice è la possibilità per il programmatore di ampliare facilmente le capacità di calcolo, per mezzo della definizione di **nuove** funzioni.

Calcolare l'area di un cerchio in CaML

```
# let pi = 3.1415927;;
pi : float = 3.1415927

# let radius = 2.0;;
radius : float = 2.0

# let area = radius *. radius *. pi;;
area : float = 12.56
```

Questa è la trascrizione di una sessione interattiva (il linguaggio in questo caso è interpretato) in cui le linee precedute da # sono scritte dall'utente mentre le altre sono il risultato della valutazione di quanto scritto nella riga precedente.

Calcolare il quadrato di un numero in CaML

```
# let square x = x * x;;
val square : int -> int = <fun>

# square 120;;
- : int = 14400
```

In questo caso viene definita una funzione che calcola il quadrato di un numero intero e, dopo la definizione della funzione, viene evidenziato il fatto che `square` è una funzione, quale sia il suo dominio e il suo codominio. Quando si applica a un parametro si calcola il risultato numerico.

Il processo di trasformazione dei dati in ingresso nei risultati avviene tramite una successione di chiamate a funzioni, il cui risultato viene passato alla funzione successiva sino ad ottenere il risultato finale.

2.4.5 Commenti

In qualsiasi linguaggio di programmazione i commenti sono la porzione di codice che non viene utilizzata dal compilatore, ma è destinata all'interpretazione da parte delle persone che leggono il codice. Proprio perché in C c'è una grande facilità a generare codice poco comprensibile, è ancora più importante che in altri linguaggi commentare adeguatamente il codice. Nel caso di codice prodotto

a fini didattici, la mancanza o la scarsezza di commenti sono considerati come **errore grave**, al pari di difetti evidenti nel funzionamento del codice.

In questo corso useremo le seguenti regole, per definire la corretta utilizzazione di commenti:

1. Ogni programma ed ogni funzione al suo interno deve avere un commento che documenti almeno chi ne sia l'autore, quando è stato realizzato e perché, quali sono i parametri in ingresso e che risultato si ottiene eseguendolo;
2. Tutte le dichiarazioni devono essere accompagnate da commenti che spieghino l'uso che verrà fatto di ciò che è stato dichiarato (variabili e quant'altro);
3. Ogni passaggio di calcolo che non è **palesemente** auto-esplicativo deve essere commentato.

Come regola di buon senso si può sempre pensare che la quantità di commenti all'interno di un programma sia almeno pari alla quantità di codice propriamente detto e, nel caso di un principiante, questo rapporto aumenta!

Formato dei commenti

Commento posto all'inizio del programma o di una funzione:

```
/*-------------------------------------------------------------
| Autore: Lionel Messi
| Data: 04 ottobre 2013
| Scopo: questo programma calcola la probabilita' p che
|  un pallone calciato da un numero n di metri dalla porta
|  avversaria entri in rete
| Input: n (reale)
| Output: p (reale)
-------------------------------------------------------------*/
```

Commento a corredo della dichiarazione di una variabile:

```
int wall; // numero di avversari in barriera
```

Commento esplicativo di un passaggio rilevante:

```
/* Il numero di avversari in barriera viene moltiplicato per
 * il valore del portiere ottenuto dall'annuario del
 * calcio Sandwich per ottenere il valore totale di difesa */

defence = wall * gk;
```

Capitolo 3

Le strutture del linguaggio

3.1 Operatori

In un linguaggio di programmazione è importante avere la possibilità di svolgere le operazioni matematiche elementari. Per questo sono definiti, in ogni linguaggio di programmazione, gli operatori che consentono di effettuare tali operazioni. Almeno le operazioni aritmetiche di addizione, sottrazione, moltiplicazione e divisione, sono disponibili praticamente in ogni linguaggio e i simboli utilizzati per descriverle sono molto spesso i medesimi simboli che si utilizzano in aritmetica.

3.1.1 Operatore e operandi: posizioni reciproche

Un prima classificazione degli operatori può essere basata sulla posizione reciproca di operatore e operandi, che non necessariamente deve essere quella tradizionale dell'operatore frapposto tra i due operandi (nel caso di operatori binari). Si possono avere tre diversi casi.

Infissi È il caso più comune: l'operatore si trova **tra** gli operandi

`5 + 3` è il normale stile per indicare addizione

Prefissi In questo caso l'operatore **precede** il o gli operandi

`-6` davanti all'operando opera un'inversione di segno
`sqrt(9.0)` in termini semantici è un operatore prefisso

Postfissi Gli operatori postfissi **seguono** il o gli operandi

`40 60 add 2 div` è un frammento di codice PostScript

Un linguaggio con operatori prefissi: Lisp

Lisp è il secondo più vecchio[1] linguaggio di programmazione ad alto livello, la sua progettazione iniziale risale al 1958. Nella definizione della grammatica di Lisp si era adottata una strategia molto lineare, tutti gli operatori sono prefissi con la medesima sintassi che fa uso di parentesi (***op* `a b`**).

`(* (+ 1 3) 2)` calcola la somma di `1` e `3` e la moltiplica per `2`

Questa notazione è nota come **notazione polacca** dato che è stata introdotta, negli anni '20 del novecento, dal matematico polacco Łukasiewicz al fine di rendere la descrizione delle operazioni, binarie o meno, non ambigua.

```
5 - 6 * 7[a]
* - 5 6 7[b]
```

[a] è una notazione ambigua, non sappiamo se dobbiamo seguire la regola di base che suggerirebbe di eseguire prima l'operazione più a sinistra, oppure se dare la precedenza all'operatore di moltiplicazione

[b] è una notazione non ambigua: si associa **sempre** ogni operatore ai due operandi che lo seguono

Un linguaggio con operatori postfissi: PostScript

Il linguaggio PostScript nasce negli anni '80 come linguaggio di descrizione di pagina, ovvero per permettere di descrivere, in maniera compatta, la posizione di testo, grafica e immagini su una pagina stampata. Dalla sua introduzione sono stati sviluppati interpreti PostScript che sono utilizzati dalle stampanti.

Il seguente frammento di codice:

```
(Hello!) 100 200 moveto show
```

sposta la posizione di stampa (istruzione `moveto`) nel punto di coordinate `(100,200)` e stampa (istruzione `show`) "`Hello!`".

Al pari di Lisp, PostScript è un linguaggio interpretato. Dovrebbe essere evidente come sia più semplice da scrivere l'interprete se si utilizza una notazione non ambigua: si risparmia molta fatica nella definizione delle precedenze che consente di semplificare il meccanismo di interpretazione.

Il linguaggio è stato sviluppato in ambito aziendale, dalla società Adobe, che ne ha poi riutilizzato una larga parte incorporandolo come *engine* di interpretazione del layout all'interno del più diffuso formato per la descrizione di documenti: *Page Description Format* (PDF).

[1] Il linguaggio considerato più vecchio è Fortran, per il semplice fatto che è il primo linguaggio per cui è stato scritto un compilatore.

3.1.2 Operatori aritmetici

Le quattro operazioni

I principali operatori che si utilizzano in ogni linguaggio di programmazione, C compreso, sono gli operatori che servono per eseguire le operazioni aritmetiche:

Operazione	Operatore	Posizione
Addizione	`+`	infisso
Sottrazione	`-`	infisso
Moltiplicazione	`*`	infisso
Divisione	`/`	infisso
Modulo (resto della divisione)	`%`	infisso
Inversione	`-`	prefisso

Attenzione ai tipi L'operatore `%` opera solo su operandi interi: $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. Non è, quindi, difficile definire sia il tipo degli operatori che utilizza, sia il tipo del risultato che genera.

Nella maggior parte dei linguaggi, però, tutti gli altri operatori operano sia su operandi interi che reali e, in effetti, dal punto di vista della macchina, ognuno di essi è ***overloaded***: indica, con uno stesso simbolo, due diverse operazioni: una con operandi e risultato interi ($\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$) e una con operandi e risultato reali ($\mathbb{R} \times \mathbb{R} \to \mathbb{R}$)

Solo nel caso della divisione questa distinzione è evidente dal risultato

```
int a, b, c;
a = 10;
b = 4;
c = a/b;
```

Per effetto dell'esecuzione dell'ultima istruzione, nell'esempio precedente, alla variabile `c` viene assegnato il valore `2`, ovvero il quoziente della divisione intera tra gli operandi. Se avessimo effettuato una divisione tra reali il risultato dell'operazione sarebbe stato diverso (`2.5`).

Divisione in C: lo zero come divisore Come si deve comportare il nostro **ambiente**[2] nel caso in cui si tenti una divisione per zero? La scelta più corretta sarebbe di non consentire di effettuare un'operazione del genere, in quanto il risultato non è un numero rappresentabile (matematicamente il risultato si indica con il simbolo ∞ e si legge infinito).

[2]Per ambiente intendiamo il complesso di linguaggio, compilatore, linker e supporto a run-time

Nel caso del linguaggio C la **cattiva notizia** è che il linguaggio non prevede un comportamento standard in questo caso, mentre la **bella notizia** è che l'insieme compilatore (**gcc**), ambiente di sviluppo, che usiamo in laboratorio all'interno di questo corso controlla la divisione per zero già in compilazione (con un **warning**) e interrompe l'esecuzione con un'eccezione aritmetica se l'istruzione viene effettivamente eseguita

Divisione in C: un numero negativo come divisore Un altro caso particolare riguarda la divisione intera quando uno degli operandi è negativo

```
int i, j, k;
i = -9;
j = 7;
k = i/j;
```

Per lo standard C89 `k` può valere sia `-2` che `-1` (arrotondamento per difetto o per eccesso), mentre in C99 il risultato è sempre arrotondato verso lo 0 (quindi il risultato è sempre `-1`). Questo è un buon esempio di indeterminazione del linguaggio migliorata nel passaggio da uno standard al successivo.

Modulo: un numero negativo come divisore Anche nel caso dell'operazione modulo, se uno degli operandi è negativo il risultato era indefinito con lo standard C89

```
int i, j, k;
i = -9;
j = 7;
k = i%j;
```

Ancora una volta, per lo standard C89 `k` può valere sia `-2` che `5` a seconda dell'implementazione, mentre in C99 il risultato è sempre `-2`.

3.1.3 Precedenza degli operatori

Regole di precedenza

Abbiamo visto che una brillante soluzione al problema di come si determina la sequenza delle operazioni da eseguire è costituita dall'uso di operatori prefissi e postfissi. La regola di valutazione dell'espressione, usando questo tipo di operatori, è molto semplice da descrivere: nel caso di operatori prefissi si prendono come operandi i due che seguono, mentre nel caso di operatori postfissi si prendono come operandi i due che precedono. In entrambi i casi, la mancanza del numero sufficiente di operandi determina un errore

Se si utilizzano operatori infissi abbiamo la necessità di definire delle **regole di precedenza** per evitare di ottenere risultati non corretti:

```
int i;
i = 8 - 3 * 2;
```

Ci aspettiamo che il risultato della valutazione dell'espressione che sta sul lato destro dell'operatore di assegnamento[3] corrisponda a quanto si calcolerebbe utilizzando le normali regole dall'aritmetica quindi sia `2`.

Eseguendo le operazioni in sequenza otterremmo come risultato `10` che, invece, è scorretto, sempre secondo le regole dell'aritmetica, che ci dicono che, in un'espressione, **prima** si eseguono le moltiplicazioni e **poi** le addizioni.

Una soluzione a questo problema consiste nell'inserire le regole di precedenza aritmetiche all'interno del modo di operare del compilatore (o dell'interprete). Il linguaggio C, come praticamente tutti gli altri linguaggi che utilizzano operatori infissi, definisce le seguenti precedenze tra operatori aritmetici (dalla più alta,1, alla più bassa, 3):

1	– prefisso	+ prefisso
2	`*` `/` `%`	
3	– infisso	+ infisso

Alterazione delle regole di precedenza Anche in questo caso il linguaggio di programmazione mutua la sintassi del linguaggio matematico: per variare l'ordine di precedenza nell'esecuzione delle operazioni si utilizzano parentesi (sempre tonde!) per raggruppare le operazioni. Il compilatore è guidato nella sua analisi di un'istruzione alla volta dalle regole di precedenza e dal raggruppamento con le parentesi.

```
i = 8 - 3 * 2;[a]
i = (8 - 3) * 2;[b]
```

[a] prima si esegue `3 * 2` → `6`, poi si esegue `8 - 6` → `2`
[b] prima si esegue `8 - 3` → `5`, poi si esegue `5 * 2` → `10`

La regola per l'interpretazione della precedenza è quindi semplice: quando due o più operatori appaiono nella stessa espressione si possono pensare come se le sottoespressioni fossero raggruppate con parentesi, con livello decrescente di precedenza, da sinistra verso destra[4].

`i + j * k`	equivale a	`i + (j * k)`
`-i + -j`	equivale a	`(-i) + (-j)`
`+i + j/k`	equivale a	`(+i) + (j/k)`

[3] D'ora in poi lo chiameremo molto spesso *rvalue*, che sta per *right value*, ovvero valore a destra dell'operatore di assegnamento.

[4] Si può sempre utilizzare l'ordine da sinistra a destra sfruttando l'associatività di tutte le operazioni binarie coinvolte.

In Fortran, linguaggio in cui si a ha disposizione l'operatore `**` per l'elevazione a potenza, l'espressione $a + \frac{b \cdot c^{d^e}}{f}$ si scrive:

`a + b * c**d**e/f` che equivale a `a + ((b * (c**(d**e)))/f)`

Il caso particolare del C C (e i suoi discendenti) ha una struttura della precedenza tra operatori molto più complessa della maggior parte degli altri linguaggi. Considerando tutti gli operatori si hanno almeno **15** diversi livelli di precedenza, paragonati agli **8** di Fortran, i **6** di Ada, i **4** di Pascal e **1** di Smalltalk e APL in cui si devono obbligatoriamente utilizzare le parentesi per raggruppare le operazioni.

Questa scelta è dovuta alla volontà di sanare gli errori formali di progettazione del linguaggio forzando le precedenze. Sarebbe stato ovviamente più efficace basarsi su un maggiore rigore di implementazione ma, come abbiamo visto, il linguaggio non è originato da un progetto accademico ma dalla necessità di disporre, operativamente, di uno strumento di sviluppo in tempi anche relativamente brevi.

La conseguenza è che è **molto difficile** ricordarsi tutte le precedenze del linguaggio ed è quindi fortemente consigliato di controllare **sempre** il manuale di riferimento!

3.1.4 Associatività

Parlando di operatori aritmetici abbiamo dato per scontato che la regola di associatività sia da sinistra a destra (*left associative*). Significa che se, nella stessa espressione, ci sono due o più operandi allo stesso livello di precedenza si eseguono, nell'ordine, dal più a sinistra al più a destra:

`i + j - k + m` equivale a `((i + j) - k) + m`

Gli operatori prefissi sono invece associativi da destra a sinistra (*right associative*) e quindi:

`+ -k` equivale a `+ (-k)`

Un curioso esempio di operatore infisso associativo a destra è ancora una volta l'elevazione a potenza in Fortran:

`4**3**2` vale `262144` e non `4096`

3.2 Assegnamento

L'assegnamento di valori a variabili è il principale meccanismo per la modifica dell'ambiente di esecuzione di un programma nel paradigma imperativo.

3.2.1 Assegnamento semplice

Cosa vuol dire eseguire un programma? In un linguaggio imperativo/procedurale come il C, la computazione si traduce, in prima approssimazione, in una serie ordinata di cambiamenti delle variabili contenute nella memoria a disposizione del programma. Gli assegnamenti forniscono il metodo principale per effettuare queste modifiche. L'effetto della valutazione fatta dall'operatore di assegnamento (= in C) è quello di calcolare il valore di un'espressione **e** e copiarlo in una variabile **v** con **e** e **v** che si trovano, rispettivamente, sul lato destro e sinistro dell'operatore: $\mathbf{v} = \mathbf{e}$. Per questo **v** si definisce *left value* (*lvalue*) e **e** si definisce *right value* (*rvalue*).

Tipi di rvalue L'espressione **e** può essere più o meno semplice da valutare per ottenere un valore da assegnare a **v**.

1. Può essere una costante e questo non pone problemi di valutazione, dato che il valore della costante è sempre lo stesso in qualsiasi contesto:

   `i = 5;` dopo l'esecuzione `i` vale **5**

2. Può essere una variabile e in questo caso è solo richiesto di recuperare il valore della variabile dalla locazione di memoria in cui è memorizzata:

   `j = i;` dopo l'esecuzione `j` vale **5**

3. Può essere un'espressione più complessa che deve essere valutata con le regole opportune e il cui valore viene poi utilizzato per l'assegnamento:

   `k = 10 * i + j;` dopo l'esecuzione `k` vale **55**

   In questo particolare esempio vengono eseguite, nell'ordine, la moltiplicazione, l'addizione e infine l'assegnamento.

L'assegnamento come operatore in C

Al contrario di quasi tutti gli altri linguaggi di programmazione in cui l'assegnamento è **un'istruzione** in C l'assegnamento è **un operatore**. Mentre un'operazione ha lo scopo principale di generare un valore, l'istruzione ha invece come scopo principale quello di modificare l'ambiente complessivo in cui il programma opera, cioè l'insieme di valori che assumono le variabili e gli altri oggetti ad esse simili. Parrebbe quindi scontato di pensare all'assegnamento come a un'istruzione.

Questa strana scelta di progetto del linguaggio consente però di poter scrivere nel programma assegnamenti a catena:

```
int i, j, k;
i = j = k = 0;
```

ha l'effetto di assegnare a tutte e tre le variabili il valore **0**. L'assegnamento è associativo a destra, quindi la semantica del codice precedente deve essere accuratamente considerata e spiegata:

1. Si esegue l'operazione `k = 0` che, oltre a modificare `k`, valuta `k` e dà come risultato `k` stesso
2. Si esegue l'operazione `j = k` (`k` è il risultato della valutazione del *rvalue*) che dà come risultato `j`
3. Si esegue l'operazione `i = j` che dà come risultato `i`

Se volessimo raggruppare esplicitamente le operazioni compiute utilizzando parentesi e mantenendo lo stesso ordine di esecuzione dovremmo scrivere:

```
i = (j = (k = 0));
```

Questa scelta dei progettisti del linguaggio, che consente di accelerare l'esecuzione di una serie di assegnamenti in cascata, può purtroppo avere delle terribili conseguenze sulla chiarezza del codice:

```
int i, j, k;
i = 1; j = 2; k = 3;
i = j * (k = i);
```

Quanto vale **i** al termine dell'esecuzione?
La semantica del codice precedente è

1. Si esegue l'operazione `k = i` che dà come risultato `k` che, dopo la valutazione, vale `1`
2. Si esegue l'operazione `j * k` che dà come risultato `2`
3. Si esegue l'operazione `i = 2` che dà come risultato `i`, quindi `2`

È assolutamente sconsigliabile usare costrutti di questo tipo!

3.2.2 Il side effect

Si dice che un costrutto di un linguaggio di programmazione ha un *side effect* quando influenza la computazione seguente in modo diverso dal solo restituire il valore che è chiamato a calcolare (se è un'operazione) o eseguire la modifica dell'ambiente (se è un'istruzione).

L'assegnamento in C **è** il caso più eclatante di uso ed abuso di *side effect*: ciò che più interessa non è tanto il valore che viene calcolato, che nella maggior parte dei casi viene ignorato, quanto l'effetto secondario di modificare il contenuto della variabile.

Non tutti i linguaggi hanno *side effect*, ad esempio i linguaggi **puramente** funzionali come Haskell sono completamente privi di *side effect* e seguono un principio di esecuzione molto più formale ed elegante: la valutazione di una funzione dà sempre gli stessi risultati in qualunque istante della computazione.

Nei linguaggi, e sono la stragrande maggioranza, che utilizzano il concetto di variabile, invece, il valore di una variabile si modifica e influenza la computazione.

3.2.3 Gli *lvalue*

L'operatore di assegnamento è un operatore molto particolare, poiché uno dei suoi operandi deve essere **speciale**. Sul lato sinistro dell'operatore di assegnamento si può trovare solo un **lvalue**, ovvero un oggetto di tipo contenitore. Per ora conosciamo solo le variabili che hanno questa caratteristica.

Non possiamo quindi usare come *lvalue* costanti o risultati di espressioni. I seguenti sono esempi di utilizzo errato dell'operatore di assegnamento:

`12 = i;`	`12` non è un *lvalue*
`i + j = 0;`	la somma dei **contenuti** di `i` e `j` non è un *lvalue*
`-i = j;`	l'inverso di `i` non è un *lvalue*

Inizializzazione

I linguaggi imperativi forniscono il costrutto di assegnamento per modificare il valore delle variabili (più in generale degli *lvalue*) ma non necessariamente forniscono degli strumenti, espliciti o impliciti, per provvedere obbligatoriamente alla loro inizializzazione, ovvero all'assegnazione di un valore standard che possiedono **prima** di effettuare qualsiasi assegnamento.

Ci sarebbero però almeno due buoni motivi per fornire un tale meccanismo: il valore iniziale potrebbe essere allocato dal compilatore risparmiando il lavoro al supporto a run-time; usare accidentalmente una variabile non inizializzata è uno dei più frequenti errori di programmazione.

La conseguenza è che è sempre consigliabile provvedere a inizializzare le variabili, prima di ogni loro utilizzo.

Alla fine dell'esecuzione della piccola porzione di codice contenuta nell'esempio seguente:

```
int i, j, k;
k = j + i;
```

quanto vale **k**? `8470478`![5]

Il linguaggio C, fortunatamente, consente di inizializzare le variabili al momento della loro dichiarazione:

```
int i = 0, j = 36, k = 23;
```

È buona norma di programmazione provvedere sempre alla inizializzazione delle variabili, nella peggiore delle ipotesi sono alcuni caratteri sprecati, nella migliore si evita di introdurre errori difficili da identificare. Tra l'altro l'inizializzazione è a costo completamente nullo per l'esecuzione del programma.

3.2.4 Operatori di assegnamento composto

Programmando con linguaggi imperativi, come C, che si basano molto sui *side effect* sorge molto spesso l'esigenza di **aggiornare** il valore delle variabili. Molto frequentemente si effettuano operazioni quali:

```
a = a + 2;
```

Per una pura opera di semplificazione sintattica operazioni come questa e per migliorare l'efficienza del codice nel caso in cui si utilizzino altri tipi di *lvalue*, si introducono, in C, gli **operatori di assegnamento composto** che consentono di raggruppare le due operazioni precedenti in una sola:

```
a += 2;
```

C'è un operatore di assegnamento composto per ciascuna delle cinque operazioni aritmetiche (**v op e**):

op	Semantica
+=	somma **v** ad **e**, memorizza il risultato in **v**
−=	sottrae **e** da v, memorizza il risultato in **v**
***=**	moltiplica **v** per **e**, memorizza il risultato in **v**
/=	divide **v** per **e**, memorizza il risultato in **v**
%=	calcola il resto della divisione di **v** per **e**, memorizza il risultato in **v**

Le due opzioni (assegnamento più utilizzo dell'operatore oppure assegnamento composto) **sembrano** equivalenti ma hanno delle sottigliezze che le distinguono, la più evidente è che con un assegnamento composto si effettua un solo accesso in memoria anziché due. Dal nostro punto di vista, comunque, le possiamo sostanzialmente considerare equivalenti.

Questi operatori associano come l'assegnamento quindi:

`i += j += k` si legge come `i += (j += k)`

[5] Usando Code::Blocks (gcc).

3.2.5 Operatori di incremento e decremento

Due tra le più comuni operazioni che si possono effettuare su una variabile sono l'incremento e il decremento, ovvero sommare o sottrarre uno al valore della variabile. Se volessimo aggiungere o togliere uno dal valore corrente di una variabile potremmo scrivere:

```
i = i + 1;
j = j - 1;
```

Gli assegnamenti composti ci permettono di risparmiare nella scrittura di queste istruzioni:

```
i += 1;
j -= 1;
```

Il C mette a disposizione una versione ancora più compatta con gli specifici operatori di **incremento** e **decremento**:

```
i++;
j--;
```

Gli operatori di incremento e decremento appaiono banali da utilizzare ma bisogna fare attenzione ad un'ulteriore complicazione che è data dalle due diverse forme che possono assumere: operatori prefissi e postfissi.

I *side effect* delle due forme sono identici, mentre differiscono i valori restituiti:

```
int i = 1;
++i;
```

in questo caso il valore dell'espressione è `2` e il *side effect* modifica il valore della variabile `i` che da `1` diventa `2`.

Se invece scriviamo:

```
int i = 1;
i++;
```

il valore dell'espressione è `1` mentre il *side effect* rimane il medesimo: `i` passa da `1` a `2`.

Costruendo un esempio più complesso, quanto valgono `i`, `j` e `k` al termine della valutazione dell'assegnamento?

```
int i = 1, j = 2, k;
k = ++i + j++;
```

L'espressione *rvalue* dell'assegnamento si valuta nel seguente modo, tenendo conto della precedenza degli operatori e di come associano:

1. Si valuta `j++` che dà come risultato `2`;
2. Si valuta `++i` che dà come risultato `2`;
3. Si sommano i valori ottenuti e si ottiene come risultato `4`;
4. Si valuta l'assegnamento che dà come risultato `4`.

Al termine della valutazione complessiva dell'espressione (compreso l'assegnamento), per i *side effect* delle operazioni effettuate le variabili hanno i seguenti valori: `i` vale `2`, `j` vale `3` e `k` vale `4`.

Ricostruzione della computazione

Vediamo un esempio di come si possa rappresentare l'evolversi di una computazione tramite lo stato del sistema. Lo stato, in prima approssimazione, può essere dato dall'insieme di valori delle variabili che sono utilizzate in un momento dato. Ad ogni passo della computazione si aggiorna il valore di ogni variabile per aggiornare lo stato.

Prima di aver eseguito qualsiasi operazione tutte le variabili (al tempo T_0) hanno valore indefinito.

```
1 int i = 2, j = 4, k = 7;
2 i = ++j - 2;
3 k = 10 * i-- + j;
```

Variabile	T_0	T_1	T_2	T_3	
i	undef				
j	undef				
k	undef				

Dopo aver dichiarato (e inizializzato) alla riga 1, le variabili da utilizzare si ha un primo stato consistente della computazione, al tempo T_1.

```
1 int i = 2, j = 4, k = 7;
2 i = ++j - 2;
3 k = 10 * i-- + j;
```

Variabile	T_0	T_1	T_2	T_3	
i	undef	2			
j	undef	4			
k	undef	7			

Con l'inizio vero e proprio della computazione, a riga 2, si eseguono le operazioni di sottrazione, incremento e assegnamento che modificano le variabili `i` e `j`. Siamo al tempo T_2.

```
1 int i = 2, j = 4, k = 7;
2 i = ++j - 2;
3 k = 10 * i-- + j;
```

Variabile	T_0	T_1	T_2	T_3	
i	undef	2	3		
j	undef	4	5		
k	undef	7	7		

Nella riga 3 si eseguono altre quattro operazioni, nell'ordine: una moltiplicazione, un decremento, un'addizione e un assegnamento, che modificano le variabili `i` e `k`. Siamo al tempo T_3 che rappresenta anche la fine di questa semplice computazione.

```
1 int i = 2, j = 4, k = 7;
2 i = ++j - 2;
3 k = 10 * i-- + j;
```

Variabile	T_0	T_1	T_2	T_3
i	undef	2	3	2
j	undef	4	5	5
k	undef	7	7	35

3.3 Flusso di controllo

3.3.1 I costrutti di controllo del flusso

Avanti il prossimo L'ordine in cui si eseguono le operazioni è fondamentale nella determinazione dei risultati in (quasi) tutti i modelli di computazione, ci dice quale costrutto deve essere eseguito per primo, quale per secondo e così via, al fine di realizzare il compito che ci siamo dati. L'introduzione delle istruzioni di controllo del flusso è stato il più importante passo in avanti nell'evoluzione della programmazione.

In assembly, infatti, l'unica possibilità a disposizione del programmatore per variare il flusso di controllo è quella di utilizzare le istruzioni di salto (condizionato e incondizionato) ch portano molto velocemente a generare codice decisamente incomprensibile; è ciò che spesso si chiama effetto *spaghetti-code*, per analogia con un piatto di spaghetti, in cui è molto difficile seguire quale sia l'inizio e la fine di ciascun spaghetto, così come in un codice di questo genere è difficile capire quale sia l'inizio e la fine di ciascuna porzione di codice.

Costrutti base

I costrutti base per il controllo del flusso sono:

Sequenza I comandi vengono eseguiti in un **ordine specificato**, tipicamente quello in cui sono raggiunti scorrendo il programma dall'inizio alla fine; l'ordine è l'ordine standard in cui si legge un testo scritto;

Selezione Sulla base della verifica di **condizioni** effettuata a run-time si effettua una scelta tra due o più **alternative** e una ed una sola viene eseguita;

Iterazione Una certa porzione di codice è **ripetuta**, a seconda del costrutto, un numero fisso (deciso a compile-time) oppure e variabile e calcolato in esecuzione (deciso a run-time) di volte.

Costrutti che invocano subroutine

Sono costrutti per il controllo del flusso basati sull'invocazione di subroutine:

Procedure Una **serie complessa di costrutti** è incapsulata in una subroutine che viene trattata come una **singola istruzione**, di solito con l'associazione di parametri;

Ricorsione Un'espressione è definita in termini di (una versione modificata di) **se stessa**, è un modello computazionale complesso che richiede una struttura dati apposita di controllo.

Entrambi questi meccanismi verranno illustrati nel seguito.

Costrutti avanzati

Sono costrutti più avanzati per il controllo del flusso, che **non tratteremo** all'interno di queste dispense:

Concorrenza Due o più frammenti di codice vengono eseguiti **contemporaneamente** su diversi processori o sullo stesso che dedica porzioni di tempo in alternanza ai vari frammenti.

Eccezioni Una porzione di codice viene eseguita assumendo che una serie di condizioni siano verificate, se ciò non è vero si solleva **un'eccezione** e l'ambiente a run-time passa il controllo all'apposito gestore.

Non-determinismo Non esiste un ordine predefinito, l'ordine di esecuzione viene lasciato al caso o ad una scelta determinata da fattori non controllati dal programmatore.

Event-driven

Un discorso a parte lo merita la computazione *event-driven*, in cui è l'utente, tramite l'interfaccia uomo-macchina, che determina l'ordine della computazione. Tecnicamente questo modello di computazione è **non-deterministico**, poiché il programmatore non può decidere quale sarà, a run-time, il flusso completo e ordinato del programma che ha scritto. In questo caso, però, ogni singola porzione che viene programmata deve essere considerata separatamente e si devono prendere in considerazione le variazioni globali dello stato del sistema che possono essere effettuate durante la computazione.

Il controllo in C

Nel linguaggio C i costrutti di controllo sono istruzioni e, nel caso della sequenza, la punteggiatura che riveste un ruolo estremamente importante:

Sequenza L'ordine in cui le istruzioni vengono eseguite è l'ordine in cui sono state inserite all'interno del programma, due istruzioni sono separate dal carattere `;`, sono istruzioni esplicite per la variazione della sequenza `goto` e `return`;

Selezione Le istruzioni che realizzano il costrutto di selezione sono `if` e `switch`; si distinguono per il numero di casi alternativi che possono trattare: con `if` solo due, con `switch` un numero qualsiasi;

Iterazione Le iterazioni sono effettuate utilizzando le istruzioni `for`, nel caso in cui si conosca a priori il numero di volte che si vuole ripetere il costrutto; `while` e `do` nel caso in cui si voglia definire a run-time verificando una condizione logica.

3.4 Espressioni logiche

Prima di iniziare a descrivere il funzionamento delle istruzioni di selezione dobbiamo capire quale struttura del linguaggio si utilizza per verificare le condizioni. Dobbiamo introdurre una nuova categoria di espressioni: le **espressioni logiche**.

Sono valori significativi per le espressioni logiche solo i valori di verità: **vero** e **falso**. Mentre in molti linguaggi esiste un tipo apposito che rappresenta solo i valori di verità (chiamato di solito `boolean` o `bool`), in C non esiste[6] e per rappresentare i valori di verità si usano numeri interi con la convenzione che `0` indica **falso** e `1` (o qualsiasi altro numero diverso da 0) indica **vero**.

3.4.1 Gli operatori relazionali

Gli operatori relazionali, dal punto di vista ortografico, sono gli stessi che si usano in matematica:

Operatore	Descrizione
<	minore di
>	maggiore di
<=	minore o uguale di ($\leq$)
>=	maggiore o uguale di ($\geq$)

Quando si valuta un'espressione logica, dato che sintatticamente è un'espressione intera, il risultato è un intero:

1. `0` se la condizione è **falsa**

[6] in effetti è stato introdotto in C99: `_Bool`, ma è solo un intero *camuffato*.

2. `1` se la condizione è **vera**

Un esempio di computazione che contiene operatori relazionali è:

```
int i = 1, j = 2, k;
k = i < j;
```

Purtroppo quest'espressione è sintatticamente corretta e, dopo la valutazione dell'operazione di assegnamento, `k` vale `1`, senza che questo abbia il minimo senso dal punto di vista dello stato del sistema.

Gli operatori relazionali possono essere utilizzati anche per confrontare numeri reali, ma, per la definizione che abbiamo dato in precedenza, il risultato della valutazione è sempre un intero:

```
float x = 1.2, y = 2.1;
int k;
k = x < y;
```

Anche in questo caso, dopo la valutazione dell'operazione di assegnamento, `k` vale `1`.

Gli operatori di relazione hanno precedenza inferiore agli operatori aritmetici e associano a sinistra;

```
i + j < k - 1
```

equivale a

```
(i + j) < (k - 1)
```

Utilizzando operatori relazionali, è abbastanza semplice scrivere espressioni sintatticamente corrette ma prive di senso come:

```
i + (j < k) - 1
```

Caveat! Attenzione all'uso di più operatori relazionali nella medesima espressione:

```
i < j < k
```

Interpretata come nella normale notazione matematica potrebbe sembrare che controlli se il valore della variabile `j` sia compreso tra quelli delle variabili `i` e `k`. Data l'associatività a sinistra questa espressione viene invece valutata come se fosse:

```
(i < j) < k
```

Quindi la sua semantica è:

1. Valuta se `i` è minore di `j` e restituisci `0` o `1`;
2. Confronta il valore restituito con `k`.

Che è completamente diverso!

Operatori di uguaglianza

Un'altra possibile condizione da verificare per operare una selezione è quella di controllare se due valori sono, o meno, uguali. Gli **operatori di uguaglianza** differiscono, nell'ortografia, dai corrispondenti operatori matematici:

Operatore	Descrizione
==	uguale a ($=$)
!=	diverso da ($\neq$)

Nel caso dell'operatore `!=` è abbastanza naturale che in un linguaggio di programmazione non si utilizzino caratteri non disponibili sulla tastiera. Scegliere per l'operatore di uguaglianza un simbolo diverso da = è una scelta obbligata, poiché = è già stato utilizzato per l'operatore di assegnamento!

Caveat Questa scelta ha generato una delle **insidie più grandi** del linguaggio C: scambiare tra loro i due operatori.

Anche gli operatori di uguaglianza sono associativi a sinistra e le espressioni che li contengono, valutate, danno come risultato **0** oppure **1**. Hanno una precedenza **minore** degli operatori relazionali

```
j < i != j < k
```

che equivale a

```
(j < i) != (j < k)
```

dà come risultato **1** (vero) solo se `j` è minore di `i` ma non di `k` oppure se è maggiore di `i` ma non di `k`.

In effetti è una maniera abbastanza incomprensibile di esprimere la relazione matematica $i < j < k$ o $k < j < i$.

Connettivi logici

Per verificare più condizioni contemporaneamente ed ottenere un unico valore di verità si utilizzano i **connettivi logici** (operatori logici) che consentono di calcolare espressioni di verità più complesse:

Operatore	Descrizione	Semantica
!	negazione	`!`***e1*** vale `1` se ***e1*** vale `0`, vale `0` se ***e1*** vale `1`
&&	and logico	***e1***`&&`***e2*** vale `1` se ***e1*** e ***e2*** valgono entrambe `1`, `0` altrimenti
||	or logico	***e1***`||`***e2*** vale `0` se ***e1*** e ***e2*** valgono entrambe `0`, `1` altrimenti

Valutazione con corto circuito L'utilizzo dei connettivi logici fornisce un'ottima possibilità per dare un esempio degli accorgimenti che si possono usare per migliorare l'efficienza del codice agendo a livello molto basso nella programmazione. Consideriamo l'espressione:

```
(a < b) && (b < c)
```

Il risultato finale della valutazione dell'intera espressione sarà **vero** (`1` in C) se sia la prima che la seconda disuguaglianza sono verificate, quindi se la prima dà come risultato **falso** è inutile valutare la seconda. Un compilatore che realizza **valutazione corto-circuitata** traduce questo frammento di codice in modo tale che **realmente** la seconda condizione non sia valutata se la prima vale **0**.

Lo stesso ragionamento vale nel caso dell'or logico:

```
(a < b) || (b < c)
```

Si evita di valutare `b < c` se `a < b` dà come risultato `1`.

Corto circuito e side effect Vediamo anche le insidie che si nascondono dietro un meccanismo come quello del corto-circuito: è opportuno, infatti, fare grande attenzione agli effetti secondari della corto-circuitazione. Abbiamo visto che la valutazione di un'espressione è spesso legata alla modifica del valore di una variabile: è il suo *side effect*. Ci sono casi in cui il *side effect* che ci aspettiamo dovrebbe avere luogo nella parte eventualmente non eseguita della valutazione.

```
i > 0 && ++j < 0
```

Se, nell'espressione precedente `i > 0` la variabile `j` viene incrementata, altrimenti non viene incrementata. La soluzione, ovvia, è quella di non utilizzare costrutti di questo tipo, in questo caso si deve incrementare `j` **prima** di eseguire la valutazione dell'espressione.

3.5 Il controllo tramite selezione

Come abbiamo detto, effettuare una selezione tra due o più scelte, sulla base di condizioni logiche, è uno dei meccanismi di base della computazione. Basti pensare a scelte molto semplici del tipo:

"*Se il divisore è zero non effettuare la divisione*"

che fanno parte del nostro patrimonio sin dalla scuola primaria.

Costruire alberi di scelta e assegnare a ciascun ramo dell'albero l'esecuzione dell'opportuna porzione di codice è una delle abilità primarie da acquisire nella progettazione e programmazione di sistemi software.

3.5.1 Il costrutto `if`

La possibilità di scegliere tra due opzioni viene normalmente implementata con un comando che, nella stragrande maggioranza dei linguaggi di programmazione, prende il nome di `if`, che in italiano traduciamo come **se**. Intuitivamente il se è riferito alla condizione logica da testare: se è vera si dovrà scegliere una opzione, se è falsa se ne sceglierà un'altra. Le due opzioni sono tra loro mutuamente esclusive e non si può mai eseguire sia il codice relativo alla clausola di verità che quello relativo alla clausola di falsità.

L'istruzione `if` in C

L'istruzione `if`, in C, nella sua forma base ha una sintassi molto semplice

`if` (ESPRESSIONE$_{\text{ctrl}}$) ISTRUZIONE$_{\mathcal{V}}$

La semantica dell'istruzione è quella generica del costrutto di selezione con i dovuti aggiustamenti dovuti alla gestione dei tipi di C. Dato che non esiste il tipo booleano, il tipo di ESPRESSIONE$_{\text{ctrl}}$ è `int`.

Viene valutata ESPRESSIONE$_{\text{ctrl}}$, se il risultato è diverso da `0` viene eseguita ISTRUZIONE$_{\mathcal{V}}$, altrimenti se il risultato è `0` viene eseguita l'istruzione immediatamente seguente l'istruzione `if`.

Un semplice esempio è:

```
if (i > 0) k++;
```

oppure, utilizzando uno stile che aumenta la leggibilità:

```
if (i > 0)
   k++;
```

Le due porzioni di codice sono completamente equivalenti, dato che in C il carattere che indica di andare a capo non ha alcuna influenza nella computazione. In entrambi i casi verrà incrementata la variabile `k` se nella variabile `i` c'è un valore positivo.

Expression statement

Vediamo ancora un esempio di `if` semplice:

```
if (i > 0)
    j = 2;
```

Per sintassi il costrutto che troviamo dopo la verifica della condizione dovrebbe essere un'**istruzione**, invece abbiamo un'**operazione** di assegnamento, è un errore? Se così fosse il linguaggio sarebbe praticamente inutilizzabile, dato che non si potrebbe svolgere il compito più comune nei linguaggi imperativi, cioè l'assegnamento, in risposta al verificarsi di una condizione data.

La soluzione trovata dai progettisti del linguaggio C per uscire da questo vicolo cieco è l'introduzione del meccanismo noto come ***expression statement***: ogni espressione può essere utilizzata al posto di un'istruzione. Quasi sempre quando si utilizza un'operazione al posto di un'istruzione, il valore calcolato viene scartato e si utilizza solo il *side effect*. Anche in questi due casi, infatti, sia il risultato dell'operazione di incremento che il risultato dell'operazione di assegnamento non vengono utilizzati.

Uno degli errori più comuni in C

La scelta di utilizzare il simbolo = per l'operatore di assegnamento e == per l'operatore di uguaglianza ha come conseguenza la possibilità di scambiarli, per disattenzione, molto facilmente:

```
if (i == 0)
    j = 2;
```

L'intenzione del programmatore è quella di assegnare `2` alla variabile `j` solo se il valore della variabile `i` è `0`. Il frammento di codice precedente la realizza in maniera corretta. Se, per uno sbaglio molto comune, il programmatore avesse scritto:

```
if (i = 0)
    j = 2;
```

avrebbe ancora generato un costrutto sintatticamente corretto ma la cui semantica è completamente diversa: si esegue l'assegnamento di `0` alla variabile `i`, il cui risultato (`0`) è valutato come fosse un valore logico e porta a non eseguire mai l'istruzione di assegnamento di `j`.

3.5.2 Istruzioni composte

Dalla sintassi dell'istruzione `if` parrebbe che fossimo forzati ad eseguire una singola istruzione, che può essere un'espressione considerata come un'istru-

zione. Che succede se ne vogliamo eseguire più di una? Dobbiamo utilizzare un'**istruzione composta** (*compound statement*).

Una sequenza di istruzioni, separate da `;` e racchiuse dai delimitatori di blocco { e } è trattata dal compilatore come una singola istruzione e può essere utilizzata ovunque si richieda, sintatticamente, come un'istruzione

```
if (i == 0)
{
   j = 2;
   k = 3;
}
```

3.5.3 La clausola `else`

L'istruzione `if` nella sua forma completa comprende anche una clausola accessoria che definisce l'istruzione da eseguire se la condizione non si verifica, la sintassi è:

`if` (ESPRESSIONE$_{\text{ctrl}}$) ISTRUZIONE$_{\mathcal{V}}$ `else` ISTRUZIONE$_{\mathcal{F}}$

Questo è il caso in cui le due istruzioni sono **sempre** in alternativa l'una all'altra: non è **mai possibile** che vengano eseguite entrambe.

```
if (i > j)
   max = i;
else
   max = j;
```

Costrutti di selezione annidati

Per trovare il massimo tra i valori di tre variabili è molto semplice utilizzare una serie di costrutti `if` annidati:

```
if (i > j)
   if (i > k)
      max = i;
   else
      max = k;
else
   if (j > k)
      max = j;
   else
      max = k;
```

Quando, come in questo caso, siamo in presenza di più istruzioni `if` consecutive si pone il problema di costruire la corrispondenza di ogni clausola `else` alla clausola `if` alla quale è collegata. Basarsi solo sull'indentazione, ovviamente, non è un buona scelta e può portare a trarre delle conclusioni errate, come nell'esempio seguente:

```
if (i > j)
   if (i > k)
      max = i;
else
   max = k;
   else
      if (j > k)
         max = j;
   else
      max = k;
```

Non è sempre molto semplice comprendere a quale `if` corrisponda ogni `else` ed è quindi opportuno utilizzare delimitatori di blocco per aumentare la chiarezza del codice.

Riscrivendo il codice precedente con tutti i delimitatori di blocco necessari per dare la massima chiarezza al codice si ottiene:

```
if (i > j) {
   if (i > k) {
      max = i;
   } else {
      max = k;
   }
} else {
   if (j > k) {
      max = j;
   } else {
      max = k;
   }
}
```

La regola generale di matching prevede che ogni clausola `else` venga accoppiata con la clausola `if` **immediatamente precedente**.

In un caso come il seguente è fondamentale applicare la definizione di matching senza basarsi su una prima impressione per giudicare l'accoppiamento delle clausole. Sarebbe, infatti, un grave errore giudicare solo a partire dall'indentazione.

```
int i = 2, j = 4, k = 7;
if (i > 1)
   if (j > 7)
      k = i;
else
   k = j;
```

La regola di matching ci dice che la clausola `else` si accoppia con la clausola `if (j > 7)`, cioè quella immediatamente precedente. Questo per ribadire che l'indentazione è **sempre** una norma di stile e non ha niente a che vedere con la sintassi del linguaggio.

Il codice precedente, per esempio, potrebbe essere riscritto come:

```
if(i>1)if(j>7)k=i;else k=j;
```

senza nessuna modifica della funzionalità, ma con una netta compromissione della comprensibilità. Esistono strumenti di formattazione automatica come `UniversalIndentGUI` che consentono di formattare, utilizzando le norme di stile più diffuse il proprio codice scritto nei principali linguaggi di programmazione, C compreso.

Sintassi rivista

Dal punto di vista semantico, le due forme di istruzione `if` (con e senza `else`) sono equivalenti, si introduce la prima solo per comodità. Se definiamo un'istruzione particolare, l'**istruzione nulla**, (*null statement*), possiamo ridefinire il secondo costrutto riassorbendo il primo

`if` (ESPRESSIONE$_{\text{ctrl}}$) ISTRUZIONE$_{\mathcal{V}}$ `else` ISTRUZIONE$_{\mathcal{F}}$

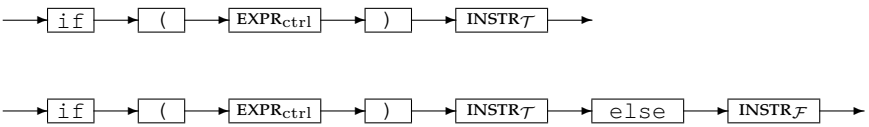
È adesso sufficiente che l'istruzione dopo la clausola `else` sia nulla per rendere questa sintassi uguale a quella del primo costrutto presentato. Ad esempio, riscrivendo il primo esempio con questa sintassi:

```
if (i == 0)
   j = 2;
else
   ;
```

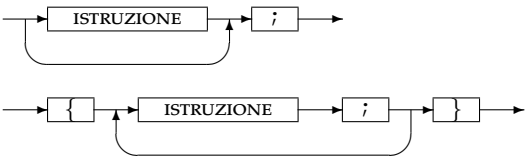
Il vantaggio della forma abbreviata (senza clausola `else`) consiste comunque nel risparmiare nella scrittura della parte di codice che non compie nessuna operazione. Inoltre neanche il compilatore deve trattare questa parte di codice.

Grafi sintattici

COSTRUTTO DI SELEZIONE



ISTRUZIONE



3.5.4 `if` in cascata

Il costrutto **if** di molti linguaggi imperativi, C compreso, è una derivazione di una notazione introdotta per la prima volta in Algol 60:

```
if CONDIZIONE then ISTRUZIONE
else if CONDIZIONE then ISTRUZIONE
else if CONDIZIONE then ISTRUZIONE
...
else ISTRUZIONE
```

Questa possibilità di impilare le varie istruzioni `if` una dietro l'altra è semplicemente il risultato dell'utilizzo di istruzioni `if` come istruzioni che seguono la parola chiave `else`. In effetti non introduce nessuna novità dal punto di vista della sintassi.

Esempio Supponiamo che i voti degli studenti universitari siano suddivisi in cinque classi di merito:

sufficiente da 18 a 20;

buono da 21 a 23;

molto buono da 24 a 26;

ottimo da 27 a 29;

eccellente 30 e 30 e lode.

Vogliamo costruire la porzione di codice che attribuisce ad una variabile `livello` un valore, crescente, da **1** a **5** a seconda del voto che si deve classificare. Risolviamo il problema scrivendo una serie di istruzioni `if` in cascata che valutano, ad ogni istruzione successiva, condizioni relative a scaglioni successivi di voto, a partire dal primo. Diamo per scontato che il valore di `voto` sia controllato in precedenza e che sia compreso tra 18 e 30.

```
if (voto <= 20)
   livello = 1;
else if (voto <= 23)
   livello = 2;
else if (voto <= 26)
   livello = 3;
else if (voto <= 29)
   livello = 4;
else
   livello = 5;
```

Proviamo ad analizzare questo frammento di codice per cercare di comprendere meglio la relazione tra le istruzioni.

Il primo `if` può essere riscritto, utilizzando una notazione più astratta del linguaggio vero e proprio, come:

```
if (voto <= 20)
    livello = 1; ← ISTRUZIONE1ramo if
else ISTRUZIONE1ramo else
```

dove ISTRUZIONE1$_{\text{ramo else}}$ è:

```
if (voto <= 23)
    livello = 2; ← ISTRUZIONE2ramo if
else ISTRUZIONE2ramo else
```

e ISTRUZIONE2$_{\text{ramo else}}$ è:

```
if (voto <= 26)
    livello = 3; ← ISTRUZIONE3ramo if
else ISTRUZIONE3ramo else
```

e infine ISTRUZIONE3$_{\text{ramo else}}$ è:

```
if (voto <= 29)
    livello = 4; ← ISTRUZIONE4ramo if
else
    livello = 5; ← ISTRUZIONE4ramo else
```

Questo `else` a chi lo do?

La sintassi dell'istruzione `if` non ci aiuta a risolvere il problema del cosiddetto ***dangling else***. Una porzione di codice come

```
if (x != 0)
if (y != 0)
   result = x/y;
else
   result = 0;
```

può essere interpretata in due modi:

1. La prima istruzione `if` è del primo tipo (senza `else`) e la seconda del secondo (con `else`);
2. La prima istruzione `if` è del secondo tipo (con `else`) e la seconda del primo (senza `else`).

Ancora una volta la regola che adotta il linguaggio è: ogni clausola `else` si accoppia sempre all'`if` più vicino che non sia già accoppiato, quindi la l'ipotesi corretta è la prima.

3.5.5 L'operatore condizionale

Oltre all'istruzione `if` il linguaggio C mette a disposizione anche un operatore con funzioni di selezione, la cui sintassi è:

$\text{ESPR}_{\text{ctrl}}$ `?` $\text{ESPR}_{\mathcal{V}}$ `:` $\text{ESPR}_{\mathcal{F}}$

Dove $\text{ESPR}_{\text{ctrl}}$ è un'espressione logica e $\text{ESPR}_{\mathcal{V}}$ e $\text{ESPR}_{\mathcal{F}}$ sono espressioni che restituiscono un tipo qualsiasi.

La semantica dell'operatore condizionale è la seguente: si valuta $\text{ESPR}_{\text{ctrl}}$, se viene valutata a **vero** (**1** o comunque un intero diverso da **0**) allora valuta $\text{ESPR}_{\mathcal{V}}$ e ritorna il suo valore come valore dell'intera espressione condizionale, se invece viene valutata a **falso** (**0**) allora valuta $\text{ESPR}_{\mathcal{F}}$ e ritorna il suo valore come valore dell'intera espressione condizionale.

Vediamo un esempio di utilizzo dell'operatore condizionale.

```
int i = 1, j = 2, k;
k = i > j ? i++ : j--;
```

Dopo l'esecuzione dell'ultima istruzione `k` vale `2`.

Ma quanto valgono `i` e `j`? Entrambe le espressioni sono valutate, oppure una sola lo è? Non è banale dare risposta a una domanda del genere e, per evitare di fare errori, è meglio usare l'equivalente istruzione `if`:

```
if (i > j)
   k = i++;
else
   k = j--;
```

3.5.6 Il costrutto `switch`

Come abbiamo visto, l'uso di costrutti `if` in cascata risolve il problema di confrontare un'espressione con una serie di valori, anziché con uno solo, e vedere con quale corrisponda. Nel caso in cui ogni condizione confronta **un'espressione intera** con una lista di **costanti** (a tempo di compilazione) allora è possibile utilizzare un altro costrutto più semplice ed efficiente, l'istruzione `switch` la cui sintassi è:

`switch` (ESPRESSIONE$_{\text{ctrl}}$) {
`case` ESPRESSIONE-COSTANTE$_1$ **:** LISTA-ISTRUZIONI$_1$
...
`case` ESPRESSIONE-COSTANTE$_n$ **:** LISTA-ISTRUZIONI$_n$
`default` **:** LISTA-ISTRUZIONI$_{\text{def}}$
}

A prima vista appare ben più complessa di quella dell'istruzione `if` ma, analizzando ogni parte dell'istruzione in dettaglio, possiamo capire che le differenze non sono così elevate. Le parti di un'istruzione `switch` sono:

ESPRESSIONE$_{\text{ctrl}}$ è un'espressione intera che viene confrontata con le costanti che sono accoppiate ad ognuna delle clausole `case`

`case` ESPRESSIONE-COSTANTE$_i$ è un'espressione che deve poter essere valutabile a tempo di compilazione, `6` lo è, `4 + 2` anche, `i + 1` no; nella lista dei casi non può apparire due o più volte la stessa costante

LISTA-ISTRUZIONI$_i$ sono le istruzioni che vengono eseguite quando ESPRESSIONE$_{\text{ctrl}}$ equivale a ESPRESSIONE-COSTANTE$_i$

`default` se ESPRESSIONE$_{\text{ctrl}}$ non equivale a nessuna delle ESPRESSIONE-COSTANTE$_i$ allora si eseguono LISTA-ISTRUZIONI$_{\text{def}}$

Proviamo adesso a progettare il frammento di codice che scrive il nome del giorno della settimana dato il numero d'ordine.

```
switch ( day ) {
   case 1: printf("Lunedì");    break;
   case 2: printf("Martedì");   break;
   case 3: printf("Mercoledì"); break;
   case 4: printf("Giovedì");   break;
```

```
    case 5: printf("Venerdì");   break;
    case 6: printf("Sabato");    break;
    case 7: printf("Domenica");  break;
    default: printf("???");      break;
}
```

Abbiamo utilizzato una nuova istruzione: `break`.

L'istruzione `break`

La semantica dell'istruzione `switch` è abbastanza lineare, dopo tutto:

1. Confronta ESPRESSIONE$_{\text{ctrl}}$ con tutte le espressioni costanti a partire da ESPRESSIONE-COSTANTE$_1$
2. Se ne trovi una, sia ESPRESSIONE-COSTANTE$_k$, uguale a ESPRESSIONE$_{\text{ctrl}}$ passa il controllo alla prima istruzione dopo `case` ESPRESSIONE-COSTANTE$_k$
3. Se non ne trovi una e esiste la clausola `default` passa il controllo alla prima istruzione dopo `default`
4. Altrimenti passa il controllo alla prima istruzione dopo l'istruzione `switch`

Se l'obiettivo del programmatore è quello di eseguire **solo** le istruzioni che sono comprese tra `case` ESPRESSIONE-COSTANTE$_k$ e `case` ESPRESSIONE-COSTANTE$_{k+1}$ allora è necessario inserire un'istruzione che passa il controllo all'istruzione immediatamente seguente il blocco corrente: questa è proprio la semantica dell'istruzione `break`.

Perché lo `switch`?

Le vere motivazioni che sono alla base dell'introduzione dell'istruzione `switch` sono legate alla maniera con cui il compilatore la tratta in confronto agli `if` in cascata. La traduzione degli `if` è costituita da una serie di etichette e goto che consentono di riprodurre, in assembly, il flusso di controllo strutturato del linguaggio ad alto livello. La traduzione dello `switch` è più semplice ed efficiente: una tabella di indirizzi sequenziali, ognuno dei quali è associato alle istruzioni che seguono un `case` che, in assembly, sono in sequenza. Questo è anche il motivo per cui è necessario inserire un'istruzione `break`, se non si vuole proseguire con le altre istruzioni della sequenza. Tutte le istruzioni `break` dentro lo `switch` sono tradotte con lo stesso salto incondizionato alla prima istruzione che segue il blocco.

Il *fall-through*

Una conseguenza di questa traduzione del costrutto `switch` come salto **precalcolato** è il comportamento nel caso in cui non mettiamo **di proposito** dei `break`:

```
switch ( month ) {
   case 11:
   case 4:
   case 6:
   case 9: ndays = 30; break;
   case 2: ndays = 28; break;
   case 1:
   case 3:
   case 5:
   case 7:
   case 8:
   case 10:
   case 12: ndays = 31; break;
   default: ndays = 0; break;
}
```

Cosa consente di calcolare questa istruzione? Il numero di giorni del mese dato il numero d'ordine del mese all'interno dell'anno, utilizzando la ben nota formula:

"Trenta dì conta novembre
con april, giugno e settembre.
Di ventotto ce n'è uno,
tutti gli altri ne han trentuno."

Nel caso in cui si abbiano dei `case` la cui lista di istruzioni è vuota si dicono *empty arm* (rami vuoti).

3.6 Il controllo tramite iterazione

L'altro meccanismo di base per modificare il flusso di controllo del programma è l'**iterazione**. Con il termine iterazione si intende la possibilità di mettere a disposizione del progettista del programma dei costrutti che permettano di eseguire la medesima porzione di codice più di una volta.

3.6.1 Tipi di iterazione

Perché programmi piccoli fanno grandi conti **Iterazione** e **ricorsione** sono i due meccanismi che permettono al computer di realizzare ripetutamente operazioni simili. Senza almeno uno di questi meccanismi, il tempo di esecuzione di un programma sarebbe una funzione lineare della dimensione del testo del programma stesso. Si avrebbe come conseguenza che il carico di lavoro che un programma potrebbe realizzare sarebbe anch'esso direttamente proporzionale

allo spazio occupato per scrivere il programma. Possiamo dire che iterazione e ricorsione sono i meccanismi che rendono veramente utili i computer.

I cicli

Nella programmazione imperativa si tende ad utilizzare molto di più il meccanismo di iterazione piuttosto che quello di ricorsione. Il meccanismo di ricorsione è invece la base su cui si fonda la programmazione funzionale. In molti linguaggi, e C è uno di questi, il meccanismo di iterazione prende la forma di **ciclo** (*loop*). Il principio di base per la definizione di un costrutto iterativo è: si deve definire una sequenza di istruzioni da eseguire un numero finito di volte, mutando, ogni volta, lo **stato dell'esecuzione**. Allo stesso modo dei comandi in una sequenza, le iterazioni in un ciclo sono eseguite generalmente per i loro effetti secondari (*side effect*): la modificazione delle variabili.

Due tipi di iterazione I cicli si presentano in due tipi che differiscono per i meccanismi utilizzati per determinare quante volte deve essere ripetuta l'iterazione:

1. Un ciclo del tipo *logically controlled* (**controllo basato su condizione logica**) viene eseguito fin tanto che una qualche condizione logica, che deve dipendere da valori modificati all'interno del ciclo, cambia valore; la **condizione logica** è lo stato della computazione;
2. Un ciclo del tipo *enumeration-controlled* (**controllo basato sul conteggio**) è eseguito una volta per ogni valore in un insieme finito dato, il numero di iterazioni è conosciuto prima che la prima iterazione inizi; il **valore numerico del controllo** è lo stato della computazione.

Nella maggior parte dei linguaggi, C compreso, questi due tipi di costrutti sono, almeno sintatticamente, distinti.

3.7 Cicli controllati da condizione logica

Uno dei modelli di computazione iterativa è quello in cui si ripete un'operazione sino a che non si verifica una condizione logica. Nella vita reale, delle istruzioni che producono un comportamento di questo tipo potrebbero essere, per esempio, le seguenti:

"Prosegui in direzione nord sino a che non ti trovi in riva al mare"

Il numero di passi che si devono fare per arrivare alla fine della computazione (in questo caso è la camminata sino al mare) non sono noti a priori, anzi, potremmo dire che rappresentano, se contati, un risultato dell'esecuzione.

3.7.1 Controllo e poi esecuzione

Controllo antecedente al ciclo I cicli controllati da condizioni logiche non sono troppo complessi da analizzare dal punto di vista semantico, l'unica vera questione da affrontare è: in quale punto del ciclo debba essere testata la condizione di terminazione? L'approccio largamente più comune è quello che prevede di testare la condizione prima di ogni iterazione. La comune sintassi `while` per questo tipo di costrutto è stata introdotta per la prima volta in Algol W:

`while` CONDIZIONE `do` ISTRUZIONE

che si legge come: fintanto che la CONDIZIONE è vera esegui l'ISTRUZIONE.

L'istruzione `while` in C

Anche in linguaggio C, l'istruzione `while` è la più semplice tra quelle che consentono di creare cicli. La sua sintassi è:

`while` `(` ESPRESSIONE$_{\text{ctrl}}$ `)` ISTRUZIONE$_{\text{loop}}$

Ancora una volta la semantica dell'istruzione deve tener conto della gestione dei tipi in C e, quindi, del fatto che non esista il tipo booleano.

Viene valutata ESPRESSIONE$_{\text{ctrl}}$, se il risultato è diverso da `0` viene eseguita ISTRUZIONE$_{\text{loop}}$, se, invece, il risultato è `0` viene eseguita l'istruzione immediatamente seguente l'istruzione `while`.

Un semplice esempio di utilizzo è il seguente:

```
int i = 0;
while (i < 10) i++;
```

in cui si incrementa una variabile (`i`) ad ogni esecuzione del ciclo, che ha come controllo di terminazione proprio la verifica del valore della stessa variabile, e si interrompe quando assume un valore superiore a `10`.

Come nel caso dell'istruzione `if`[7], anche in questo caso, nella progettazione del linguaggio, si è deciso di fare a meno di una parola chiave. L'istruzione di ciclo, infatti, segue direttamente la parentesi che chiude la condizione. Interpretando la semantica dell'istruzione ci si rende conto che ISTRUZIONE$_{\text{loop}}$ potrebbe non essere mai eseguita. Questo accade se la condizione viene valutata falsa (cioè zero) alla sua prima valutazione.

[7] Non si utilizza una parola chiave per la clausola di verità, ma si esegue direttamente l'istruzione che segue l'istruzione `if`. In molti linguaggi si utilizza una parola chiave, quasi sempre `then`.

Trovare gli n numeri la cui somma è maggiore di m Supponiamo di voler calcolare qual è il numero minimo n di numeri consecutivi, a partire da 1, che devo sommare per ottenere come somma un valore maggiore o uguale a un numero dato m. Questo problema si si presta bene ad essere risolto mediante il controllo di una condizione logica controllando, dopo ogni iterazione, se il valore della somma è minore, uguale o maggiore a m.

```
int m = 100, n = 1, sum = 0;
while (sum < m) {
   sum += n;[a]
   n++;[b]
}
```

[a] equivale a `sum = sum + n`
[b] equivale a `n = n + 1`

Dopo l'esecuzione dell'istruzione `while` ho a disposizione, nella variabile n, il valore cercato.

Quando è perché è necessario usare questo costrutto

È importante notare che l'uso dell'iterazione guidata dal controllo di una condizione logica è tipico in un caso come questo in cui non si sa **a priori** quante volte deve essere ripetuto il ciclo. In questo caso ci poniamo come obiettivo che la somma debba superare il valore di un m dato, e, ogni volta che il valore della somma varia, lo controlliamo per verificare la condizione. In effetti il numero di volte n che viene ripetuto il ciclo è proprio il **risultato** della computazione. Da notare che corrisponde a risolvere il problema di trovare il numero n più piccolo che risolve la disequazione $\frac{n \cdot (n-1)}{2} > 100$. Un metodo di risoluzione come quello che abbiamo ipotizzato è un semplice esempio di **risoluzione numerica** anziché analitica.

Il problema della terminazione

L'introduzione dei costrutti di iterazione ci propone per la prima volta un problema di cui bisognerà sempre tener conto nella progettazione di un programma: **come facciamo ad avere la sicurezza che il programma termini la propria esecuzione?** Fintanto che l'unico costrutto che consente di raggruppare le istruzioni tra loro è il costrutto sequenza questo problema non si pone affatto: si tratta di una sequenza lineare di passi che il programma deve compiere dal primo all'ultimo e, una volta raggiunto l'ultimo, interrompe la propria esecuzione. Adesso, invece, l'interruzione di una porzione di computazione è legata al falsificarsi di una condizione logica: siamo sicuri che questa condizione, prima o poi, si falsifichi?

Cicli infiniti Utilizzando l'istruzione `while` è molto semplice scrivere codice che programma cicli infiniti, che non terminano. Questo si può fare volontariamente:

```
#define TRUE 1
while (TRUE) ;
```

oppure, senza rendersene conto, per un errore di progettazione

```
int m = 100, n = 1, sum = 0;
while (sum < m) {
   n++;
}
```

In questo caso ci si dimentica banalmente di aggiornare la variabile `sum` all'interno del ciclo, con conseguenze disastrose!

Per evitare che si verificano casi di cicli infiniti dobbiamo sempre assicurarci, durante la progettazione del codice, che siano verificate due condizioni:

1. Le variabili che utilizziamo nella valutazione dell'espressione che rappresenta la condizione logica di terminazione del ciclo **devono essere aggiornate** durante l'esecuzione del ciclo stesso (ancora una volta entra in gioco il concetto di stato della computazione);
2. L'**andamento del valore** di queste variabili durante l'esecuzione del ciclo deve essere quello che ci aspettiamo, per esempio un incremento o decremento costante, e non il contrario.

Nell'esempio seguente, un errore che può essere di distrazione, eseguendo una sottrazione anziché una somma per l'aggiornamento della variabile di controllo del ciclo genera una situazione di ciclo infinito:

```
int m = 100, n = 1, sum = 0;
while (sum < m) {
   sum -= n;[a]
   n++;
}
```

[a] errato! si effettua la sottrazione anziché la somma

3.7.2 Esecuzione e poi controllo

Controllo posticipato al ciclo Può essere talvolta utile controllare la condizione logica di terminazione non all'inizio ma al termine del ciclo di istruzioni. Non tutti i linguaggi dispongono di un costrutto di questo tipo perché, come vedremo non è essenziale ai fini di una corretta progettazione dell'iterazione. Il linguaggio in cui è stato introdotto questo tipo di costrutto è il Pascal:

`repeat` ISTRUZIONE `until` CONDIZIONE

La semantica di questo costrutto è: ripeti l'ISTRUZIONE finché non si verifica la CONDIZIONE

```
repeat
   readln(line)
until line[1] = '$';
```

L'istruzione `do` in C

Il linguaggio C fornisce una condizione di controllo posticipato che funziona in maniera logicamente inversa rispetto al corrispondente comando Pascal. La sintassi dell'istruzione `do` è:

`do` ISTRUZIONE$_{\text{loop}}$ `while (` ESPRESSIONE$_{\text{ctrl}}$ `) ;`

In questo caso ISTRUZIONE$_{\text{loop}}$ viene eseguita sin tanto che ESPRESSIONE$_{\text{ctrl}}$ è **verificata**.

La sua semantica è la seguente: viene eseguita ISTRUZIONE$_{\text{loop}}$, poi viene valutata ESPRESSIONE$_{\text{ctrl}}$, se il risultato è diverso da `0` viene di nuovo eseguita ISTRUZIONE$_{\text{loop}}$, altrimenti, se il risultato è `0` viene eseguita l'istruzione immediatamente seguente l'istruzione `do`.

In ogni caso ISTRUZIONE$_{\text{loop}}$ viene eseguita almeno una volta.

`while` e `do`: confronto

L'introduzione del costrutto `do` non aggiunge nessuna evidente espressività al linguaggio. I due costrutti

`do` ISTRUZIONE$_{\text{loop}}$ `while (` ESPRESSIONE$_{\text{ctrl}}$ `) ;`

e

ISTRUZIONE$_{\text{loop}}$
`while (` ESPRESSIONE$_{\text{ctrl}}$ `)` ISTRUZIONE$_{\text{loop}}$

sono equivalenti.

Si ha un vantaggio evidente dal punto di vista della progettazione del codice solo nel caso in cui il corpo del ciclo è costituito da un numero elevato di istruzioni poiché evitare di ripetere l'insieme di istruzioni rende il programma meno soggetto ad errori.

È esattamente questo il motivo che ha fatto sì che siano state messe a disposizione entrambe le istruzioni nel linguaggio, ad un costo per il progettista, d'altronde, molto contenuto, potendo sfruttare, per l'implementazione, altri costrutti.

3.8 Cicli controllati dal conteggio

Controllare contando I meccanismi di controllo dell'iterazione basati sull'enumerazione hanno il loro capostipite nel ciclo `do` del Fortran I. Meccanismi simili sono stati adottati in qualche forma da pressoché ogni linguaggio seguente, mentre la sintassi e la semantica variano ampiamente. Anche lo stesso ciclo del Fortran si è evoluto considerevolmente nel corso del tempo.

```
do i = 1, 10, 2
   LOOP BODY
end do
```

La variabile `i` è detta **indice** del ciclo, le espressioni che seguono il simbolo di uguaglianza sono il **valore iniziale**, il **limite** ed il **valore di incremento** del ciclo:

```
do counter = 1, 5, 1
   write(*, '(i2)') counter
end do
```

Un esempio di codice Fortran che utilizza questo costrutto è il seguente:

```
   DIMENSION A(999)
   FREQUENCY 30 (2,1,10), 5(100)
   READ 1, N, (A(I), I = 1,N)
 1 FORMAT (I3/(12F6.2))
   BIGA = A(1)
 5 DO 20 I = 2,N
30 IF (BIGA-A(I)) 10,20,20
10 BIGA = A(I)
20 CONTINUE
   PRINT 2, N, BIGA
 2 FORMAT (22H1THE LARGEST OF THESE NUMBERS IS F7.2)
   STOP 77777
```

Senza scendere in particolari sulle singole istruzioni, è da notare la particolare sintassi che prevede di utilizzare un'etichetta numerica per l'ultima riga del ciclo e di avere l'etichetta come parametro dell'istruzione `DO`[8].

Utilizzando opportuni valori da assegnare al valore iniziale al limite e al valore di incremento del ciclo si possono costruire iterazioni su sequenze aritmetiche arbitrarie regolari di interi. Per ogni valore della sequenza si eseguono i calcoli opportuni e questo permette al compilatore di generare codice molto efficiente. Molti compilatori Fortran adottano addirittura la soluzione di precalcolare il numero di iterazioni ed utilizzare questo valore durante l'esecuzione

[8] Il linguaggio Fortran, al contrario di C, è *case-insensitive* e quindi le istruzioni `do` e `DO` sono equivalenti tra loro, scrivere in maiuscolo o minuscolo è solo una convenzione di stile.

del codice. Quello che **non** si può fare è costruire l'iterazione utilizzando un insieme arbitrario di numeri interi.

Sequenze arbitrarie

Per utilizzare l'iterazione utilizzando un insieme arbitrario di numeri interi si deve ricorrere al costrutto a cui tipicamente si fa riferimento come `foreach`. In `C#` esiste il comando `foreach`:

```
foreach (int x in myArray)
{
   Console.WriteLine(x);
}
```

Il comando all'interno del loop `foreach` viene ripetuto tante volte quanto vale la cardinalità dell'array `myArray` e, per ognuno dei valori contenuti, si esegue il comando (che stampa qualcosa a terminale). Vedremo come sarà possibile replicare questo comando anche in linguaggio C, che pure non lo possiede come comando predefinito.

3.8.1 Differenze nell'approccio

Quando è possibile predire il numero di operazioni da effettuare prima che si inizi ad eseguire il ciclo? Solo nel caso in cui il costrutto **serva esclusivamente** per costruire una sequenza aritmetica immutabile, non se il costrutto è un meccanismo per **rendere più semplice** l'enumerazione. Più precisamente: se il numero di iterazioni o la sequenza di valori che l'indice può assumere **possono cambiare** come risultato dell'esecuzione del ciclo allora il significato semantico del costrutto cambia radicalmente. Linguaggi quali Fortran e Ada seguono la prima strategia di definizione ed implementazione del costrutto, mentre C adotta la seconda.

3.8.2 L'istruzione `for` in C

Ad una prima analisi della sintassi, l'istruzione `for` del C sembrerebbe essere costruita solo per effettuare il controllo sulla base dell'enumerazione. La sintassi dell'istruzione `for` è:

`for` ($\text{ESPR}_{\text{init}}$; $\text{ESPR}_{\text{ctrl}}$; $\text{ESPR}_{\text{updt}}$) $\text{ISTRUZIONE}_{\text{loop}}$

Possiamo adesso scrivere la porzione di codice che calcola la somma dei primi n numeri utilizzando questo costrutto:

```
int i, n = 100, sum = 0;
for (i = 1 ; i <= n ; i++ )
   sum += i;
```

Per far sì che l'istruzione esegua strettamente il controllo basandosi sull'enumerazione dovrebbero valere una serie di condizioni:

1. Le espressioni che controllano il ciclo dovrebbero essere solo espressioni intere;
2. Nell'insieme di istruzioni che vengono eseguite durante il ciclo dovrebbe essere impedito di modificare l'indice;
3. Dovrebbe essere impedito di modificare qualsiasi variabile che serva per calcolare quando l'esecuzione si deve interrompere (condizioni di fine ciclo);
4. Non dovrebbe essere permesso né di uscire né di entrare all'interno del ciclo utilizzando altre istruzioni.

In C nessuna di queste condizioni è rispettata, per cui il costrutto `for` può assumere le connotazioni più disparate, ma non è sicuramente solo un ciclo enumerativo nel senso stretto del termine.

Espressioni non intere

Il controllo del ciclo può essere effettuato utilizzando espressioni reali.

```
float i, x;
for (i = 1.5 ; i < 3.9 ; i += .3)
   x = i*2.;
```

Questa sequenza di istruzioni, che ci si aspetta che debba essere eseguita solo nove volte, può facilmente essere eseguita dieci volte per motivi legati all'approssimazione numerica del calcolo Provando a compilare ed eseguire questa porzione di codice utilizzando il compilatore gcc si ottiene che alla decima interazione il valore dell'indice risulta essere `3.89999962` che è strettamente minore di `3.9`.

Modifica dell'indice all'interno del ciclo

Mentre i linguaggi che utilizzano il **for-loop** in maniera stretta non consentono di modificare la variabile indice all'interno del ciclo, in C questo non è controllato né dal compilatore né dall'ambiente a run-time.

```
int i, j;
for (i = 0 ; i <= 20 ; i += 2) {
   j = i*2;
   i *= 2;
}
```

Il corpo di questo ciclo, anziché essere eseguito undici volte come ci si aspetterebbe dall'analisi dell'istruzione `for`, viene eseguito solo quattro volte. Inoltre, al termine dell'esecuzione, il valore della variabile `i` è `30`, mentre ci si aspetterebbe che avesse come valore il primo numero pari superiore a venti.

Modifica della terminazione all'interno del ciclo

Oltre ad avere la possibilità di modificare la variabile indice all'interno del ciclo, è possibile, senza alcun controllo da parte né del compilatore né dell'ambiente a run-time, modificare anche le variabili che eventualmente controllano la terminazione del ciclo stesso.

```
int i, j = 20;
for (i = 0 ; i <= j ; i += 2)
   j = i+2;
```

Una situazione come quella riportata nell'esempio, in particolare, determina che l'istruzione si trasformi in un ciclo infinito.

Ingresso e uscita incondizionati dal ciclo

Una possibilità che è comune anche ad altri linguaggi, è quella di poter uscire, utilizzando istruzioni apposite, dal ciclo, ovvero passare incondizionatamente all'istruzione successiva.

```
int i, j = 20;
for (i = 0 ; i <= j ; i += 2) {
   j = i+2;
   if (j > 14)
      break;
}
```

Ancora una volta il corpo del ciclo non viene eseguito il numero di volte che si aspetterebbe dalla semplice analisi dell'istruzione `for`.

Cos'è quindi `for` in C?

Il costrutto `for` del linguaggio C **non è** l'implementazione di un'iterazione basata sull'enumerazione, ma una maniera diversa di effettuare iterazione

basata su condizioni logiche, l'istruzione `for` è un esempio del cosiddetto **zucchero sintattico**.

Utilizzare il costrutto:

```
for (i = first ; i < last ; i += step) {
    LOOP BODY
}
```

è esattamente uguale, dal punto di vista semantico, a utilizzare

```
{
    i = first;
    while (i < last) {
        LOOP BODY
        i += step
    }
}
```

Il fatto che l'istruzione `for` sia un `while` scritto in altro modo spiega anche perché si possono omettere una o più delle condizioni di controllo che dovrebbero essere presenti nell'istruzione:

1. Se si omette la dichiarazione di inizializzazione si presume che l'indice sia stato inizializzato in precedenza;

2. Se si omette il controllo di fine ciclo si assume che esso venga effettuato all'interno del ciclo stesso;

3. Se non si effettua l'update dell'indice all'interno dell'istruzione `for` è possibile farlo ancora all'interno del ciclo.

Seguendo questa logica perversa si può giungere alle estreme conseguenze di scrivere codice quale:

```
int i = 0, j;
for (;;) {
    i++;
    j = i*2;
    if (i>=10) break;
}
```

in cui, all'interno dell'istruzione `for` non si utilizza alcun elemento di quelli pensati per il controllo del ciclo.

Comma expression

A questo punto dovrebbe essere ragionevole comprendere anche perché, in inizializzazione o in controllo di fine ciclo si possono due o più espressioni anziché una, utilizzando una ***comma expression***.

$$\text{ESPR}_1 \quad , \quad \text{ESPR}_2 \quad , \quad \ldots \quad , \quad \text{ESPR}_n$$

Le varie espressioni della *comma expression*, che vengono valutate in successione, da sinistra verso destra, risultano essere, nella rivisitazione semantica del costrutto, espressioni che vengono valutate in successione rispettivamente, prima della verifica della condizione logica di iterazione, ed alla fine del corpo del ciclo.

Dal punto di vista del linguaggio una *comma expression* è un'espressione come tutte le altre, che può essere utilizzata in qualsiasi contesto in cui ci si aspetta un'espressione.

Virgola o punto e virgola? Abbiamo visto che l'elemento sintattico che separa due istruzioni è il carattere *punto e virgola* (`;`). L'elemento sintattico che, invece, separa due espressioni da valutare in sequenza per come appaiono nel programma (da sinistra verso destra) è il carattere *virgola* (`,`). Ma il linguaggio C **promuove** le espressioni al rango di istruzioni e consente di scriverle dove vorremmo poter scrivere solo istruzioni e, di conseguenza, esiste una sorta di interscambiabilità tra il simbolo di punteggiatura `;` e il simbolo `,` se le istruzioni da separare sono due espressioni interpretate come istruzioni.

```
int i, j, k;
for ( i=0 ; i<10 ; i++ ) {
   k = i*3,
   j = i*2;
}
```

Questa sintassi, pur apparendo a prima vista errata, è corretta. All'interno del corpo del ciclo `for` si esegue **una sola** istruzione che è una *comma expression* promossa a istruzione.

3.9 Tipi di dati

La maggior parte dei linguaggi di programmazione possiede la nozione di **tipo** da associare alle espressioni. L'utilizzo dei tipi ha due motivazioni principali:

1. I tipi forniscono implicitamente un contesto per molte operazioni così che il programmatore non debba specificare il contesto esplicitamente; in C, come abbiamo già visto, l'espressione `a + b` utilizzerà l'addizione tra

interi se `a` e `b` sono di tipo `int`, l'addizione tra numeri reali se `a` e `b` sono di tipo `float`;

2. L'utilizzo dei tipi limita l'insieme delle operazioni che possono essere effettuate al fine di produrre un programma semanticamente valido, ad esempio impedisce al programmatore di calcolare il coseno di una stringa di caratteri.

Un buon **sistema di tipi** all'interno del linguaggio che utilizziamo può controllare abbastanza errori da essere estremamente utile in pratica.

3.9.1 Rappresentazione dei dati

L'hardware del computer può interpretare i bit in memoria in svariate maniere differenti:

1. Istruzioni;
2. Indirizzi;
3. Caratteri;
4. Numeri interi di varia lunghezza;
5. Numeri reali di varia lunghezza.

Alla domanda: cosa rappresenta la seguente stringa di bit:

```
01100010 10111001 00101100 11100011
```

cosa si potrebbe rispondere? I bit, per loro natura, non hanno un tipo e di solito i computer non cercano di tener traccia di quali informazioni si trovano in ciascuna locazione di memoria. Diventa quindi essenziale che sia il linguaggio a tener traccia del tipo di informazioni memorizzate nelle locazioni di memoria utilizzate per il calcolo.

Confronto con l'aritmetica

In aritmetica, quando si eseguono i calcoli con carta e penna, difficilmente ci si pone il problema di effettuare un'operazione utilizzando come operandi due oggetti che sono di natura differente tra loro. Scrivere:

1.5 + 2 = 3.5

è abbastanza naturale per tutti noi e non ci porta a pensare che:

1. O stiamo sommando due numeri di natura diversa (un intero e un reale) e applichiamo una qualche regola implicita per poterlo fare;

2. O stiamo dando per scontato che il numero `2` non è in effetti un intero ma un reale scritto in maniera abbreviata che avremmo, più correttamente dovuto scrivere come `2.0`.

La situazione è più complessa e prevede una riflessione prima di operare, quando invece scriviamo

1.5 + 5/4 = ??

Adesso il problema di effettuare un'operazione tra oggetti di natura diversa si pone in pieno. Prima di effettuare la somma devo compiere una **conversione**: o riscrivo 1.5 come $\frac{6}{4}$ oppure riscrivo $\frac{5}{4}$ come 1.25. Quindi il risultato può essere $\frac{11}{4}$ se utilizzo una rappresentazione frazionaria, oppure 2.75 se uso una rappresentazione con parte intera e parte decimale. Entrambe le soluzioni sono formalmente corrette e accettabili.

3.9.2 Un sistema di tipi

Un **sistema** di tipi è costituito da:

1. Un meccanismo per definire i tipi ed associarli a certi costrutti del linguaggio;
2. Una serie di regole per determinare l'**equivalenza**, la **compatibilità**, e l'**inferenza** di tipi.

I costrutti del linguaggio che devono essere associati ad un tipo sono esattamente quelli che hanno dei valori: costanti, variabili ed altri che vedremo in seguito, nonché espressioni costruite a partire da questi costrutti.

Le regole di **equivalenza** tra tipi determinano quando i tipi di due valori sono lo stesso tipo.

Le regole di **compatibilità** determinano quando un valore di un certo tipo può essere utilizzato in un certo contesto.

Le regole di **inferenza** di tipi definiscono quando si può ricavare il tipo di un'espressione basandosi sul tipo delle parti che la costituiscono.

Regole del sistema

Le regole di **equivalenza**, ad esempio, andranno applicate per decidere se il valore valutato di una certa espressione potrà essere assegnato ad una variabile data: l'assegnamento sarà corretto solo se i due tipi sono lo stesso tipo. In caso contrario o il compilatore produce un messaggio d'errore oppure si attivano delle regole di conversione di tipo, che devono esistere nel linguaggio e che vedremo in seguito.

Le regole di **compatibilità**, ad esempio, ci dicono se posso applicare un certo operatore ad una variabile di un certo tipo: cercare di calcolare il resto della divisione tra due numeri reali utilizzando l'operatore `%` genererà un messaggio di errore perché si infrange una regola di compatibilità.

Le regole di **inferenza** sono quelli che determinano, in C, quali tra gli operatori aritmetici *overloaded* devono essere utilizzati per compiere una determinata operazione e di che tipo è il risultato ottenuto.

Type checking

Il ***type checking*** è il processo che assicura che un programma obbedisca alle regole del sistema di tipi del linguaggio adoperato. Una violazione a queste regole si dice ***type clash*** (conflitto tra tipi).

Un linguaggio si definisce **fortemente tipato** (*strongly typed*) se è proibita l'applicazione di ogni operazione ad oggetti che non è previsto che supportino quell'operazione, in un modo che consente all'implementazione del linguaggio (compilatore e ambiente a run-time) di controllare. In altre parole non ci sono meccanismo di compatibilità, per esempio non si possono effettuare operazioni tra interi e reali senza una conversione esplicita.

Un linguaggio si definisce **staticamente tipato** (*statically typed*) se è fortemente tipato e il *type checking* può essere completamente effettuato durante la compilazione.

Nel senso stretto del termine pochi linguaggi sono staticamente tipati, in pratica questo termine si applica spesso a linguaggi in cui la maggior parte del controllo dei tipi può essere effettuata durante la compilazione, e la restante (piccola) parte può essere effettuata a run-time.

Esempi di linguaggi e loro *typing*

Un esempio di linguaggio fortemente tipato è **Ada** che, tra l'altro, è in larghissima parte tipato staticamente.

Anche le implementazioni di Pascal possono effettuare la maggior parte del controllo dei tipi durante la compilazione sebbene il linguaggio non sia altrettanto strettamente tipato.

Il C nel corso della sua evoluzione ha avuto un significativo incremento del controllo di tipi, C89 è molto più fortemente tipato del linguaggio originale sebbene continui ad essere significativamente meno fortemente tipato, ad esempio, di Pascal.

Tra l'altro, le implementazioni di C, raramente effettuano dei controlli di tipo a run-time

Alcuni dei primi linguaggi ad alto livello, (es., Fortran 77, Algol60 e Basic), fornivano un insieme di tipi **piccolo**, predefinito e **non estensibile**.

Il Fortran, ad esempio, non richiede neanche che le variabili vengano dichiarate, esistono delle regole per determinare il tipo di una variabile non dichiarata

dal suo nome: tutte le variabili il cui nome inizia con una lettera compresa tra `i` ed `n` sono intere, le altre reali.

Nella maggior parte dei linguaggi correntemente in uso, tuttavia, l'utente deve dichiarare esplicitamente il tipo di ogni variabile che vuole utilizzare, ed è tenuto anche a definire, con diversi meccanismi, le caratteristiche di ogni tipo non predefinito dal linguaggio

Il C **obbliga** a dichiarare le variabili e fornisce vari meccanismi per costruire nuovi tipi di dati.

3.9.3 Tipi semplici

La terminologia per la descrizione dei tipi può variare da un linguaggio di un altro, specialmente se si tratta di linguaggi appartenenti a categorie diverse. Esistono comunque delle definizioni sufficientemente diffuse da poter essere considerate generali.

Molti linguaggi forniscono tipi predefiniti simili a quelli supportati in hardware da molti processori:

1. Valori booleani;
2. Caratteri;
3. Numeri interi;
4. Numeri reali (*floating point*).

Booleani

I valori booleani, spesso chiamati anche valori logici, sono di solito implementati con un singolo byte, in cui `1` rappresenta il valore `vero` e `0` rappresenta il valore `falso`.

Abbiamo già visto che C, in questo, è completamente fuori dalla norma perché in C manca il tipo booleano e, al suo posto, si usano numeri interi. La mancanza del tipo booleano è stata (parzialmente) rimediata nel C99 introducendo il tipo `_Bool` che, in effetti, è ancora un tipo intero in cui si è solo imposta la regola che tutti i valori diversi da zero che vengono assegnati a variabili `_Bool` sono convertiti a 1.

È un tipco esempio di scelte che si compiono per motivi di ***backward compatibility***, ovvero per mantenere corretto tutto quello che è stato fatto sino al momento in cui è stata presa questa decisione.

Caratteri

I caratteri sono stati per lungo tempo implementati utilizzando un singolo byte, tipicamente seguendo la codifica EBCDIC (ambiente IBM mainframe) o

ASCII (ambiente UNIX workstation). Linguaggi più moderni (es., Java e C#) utilizzano una rappresentazione con due byte per accogliere la porzione più comunemente utilizzata dell'insieme di caratteri Unicode.

Unicode è uno standard internazionale progettato per poter rappresentare caratteri di un'ampia varietà di alfabeti, non solo quello latino. I primi 128 caratteri di Unicode, per motivi di opportunità, sono identici ai caratteri ASCII.

In C per i caratteri si utilizza un singolo byte, si possono quindi rappresentare fino a un massimo di 256 caratteri diversi, tipicamente la cosiddetta estensione **Latin-1** di ASCII.

Tipi numerici

Praticamente in qualsiasi linguaggio di programmazione si hanno a disposizione tipi di dato che rappresentano numeri. I tipi più comuni sono **interi** e **reali**. La loro rappresentazione è strettamente legata alla modalità di rappresentazione dei numeri che si utilizza in hardware.

La più comune rappresentazione per numeri interi è il **complemento a 2**, mentre la più comune per numeri *floating point* è lo standard **IEEE 754**. Alcuni linguaggi, pochi, mettono a disposizione **diverse versioni di numeri interi e numeri reali**, Fortran e C sono tra questi.

Le varie versioni differiscono tra loro per **lunghezza**, dove il termine lunghezza è riferito al numero di bit (byte) che servono per rappresentare il numero. La maggior parte dei linguaggi, inoltre, lascia la scelta della precisione della rappresentazione all'implementazione del linguaggio. Questo può avere come conseguenza differenze in precisione tra diverse implementazioni che non consentono la completa portabilità dei programmi: programmi che si compilano ad eseguono correttamente su un sistema possono produrre errori a run-time o risultati errati su un altro.

Java e C# forniscono la soluzione migliore garantendo la presenza di tipi numerici di lunghezza variabile con precisione specificata per ognuno di essi.

Solo pochi linguaggi forniscono tipi diversi per **interi con o senza segno**, tra questi C, C++ e C#.

Pochi linguaggi (Fortran è uno di questi) mettono a disposizione un tipo predefinito per la rappresentazione dei **numeri complessi**, di solito implementati come una coppia di numeri reali, uno per la parte reale ed uno per la parte immaginaria. In C i numeri complessi sono stati introdotti nella versione C99.

Molti linguaggi di scripting supportano l'utilizzo di **interi a precisione arbitraria**, è l'implementazione ad utilizzare byte aggiuntivi ove necessario.

Interi, booleani e caratteri sono tutti esempi di tipi **discreti**: il dominio in cui sono definiti è numerabile ed è una nozione ben definita quella di predecessore e successore per ogni elemento dell'insieme.

3.9.4 I tipi semplici in C

Gli interi

I tipi interi nel C sono 6 (8 in C99) dato che si ha varietà di lunghezza e ci sono tipi per interi con e senza segno:

`short`	`unsigned short`
`int`	`unsigned int`
`long`	`unsigned long`
`long long`	`unsigned long long` (solo in C99)

Vale la pena di notare che gli interi senza segno (`unsigned`) sono i residui di un'epoca di programmazione in cui si tendeva a mettere a carico del programmatore la scelta delle soluzioni più efficienti e guadagnare anche un singolo bit ceduto dal segno alla rappresentazione del numero poteva essere utile. Linguaggi progettati in tempi più recenti come Java, infatti, non hanno gli interi senza segno

La dimensione (il numero di bit) di ciascuno dei tipi `int` è legata alla macchina su cui si opera, l'unica certezza che si ha è che

$$\texttt{short} \leq \texttt{int} \leq \texttt{long}$$

In una macchina con architettura a 32 bit `short` occupa 2 byte e `int` e `long` 4 byte, che è la massima lunghezza di parola (*word*) utilizzabile. In una macchina con architettura a 64 bit `short` occupa 2 byte, `int` 4 byte e `long` 8 byte.

La mancanza di standardizzazione sulla lunghezza dei tipi può essere fonte di errori nel porting di programmi da una macchina ad un'altra con diversa architettura (o scelta di implementazione).

I caratteri

Una conseguenza della scelta di costruire un linguaggio con scarso controllo per i tipi (*weakly typed*) è il tipo di dati che viene utilizzato per rappresentare i caratteri in C `char`. I caratteri sono interscambiabili con gli interi, la sola differenza è la loro lunghezza: un carattere è rappresentato con un singolo byte. Il numero intero contenuto nella variabile di tipo `char` è, in effetti, il numero d'ordine del carattere nella tabella di riferimento (ASCII).

L'assegnamento:

```
char c = 'a';
```

ha come conseguenza che la variabile `c` viene inizializzata con il valore `97` (decimale) poiché `a` è il 97-esimo carattere della tabella ASCII.

I numeri reali

La rappresentazione più semplice per i numeri reali è la rappresentazione *fixed-point* in cui la posizione del punto decimale è fissa nello spazio di rappresentazione. Ad esempio, un numero *fixed-point* di 8 cifre potrebbe avere il punto decimale fissato dopo la quarta cifra:

```
08234500
```

con questa convenzione la serie di cifre[9] rappresenterebbe il numero 823.45.

Il difetto di una rappresentazione come questa consiste nel non poter rappresentare, in maniera flessibile, numeri molto piccoli o numeri molto grandi. L'intervallo di rappresentazione è infatti limitato dalla dimensione dello spazio di memorizzazione.

I floating-point Questo problema si risolve utilizzando una rappresentazione basata sull'uso della notazione scientifica, mediante la quale un numero è rappresentato in una scala fissa, tipicamente tra 1 e 10, e poi scalato utilizzando una potenza di 10:

$$823.45 \rightarrow 8.2345 \times 10^2$$

Questa rappresentazione, abbastanza standard nei computer, è la cosiddetta rappresentazione *floating-point* costituita da due porzioni: **mantissa** ed **esponente**. Le cifre a disposizione vengono utilizzate in parte per le cifre significative (la mantissa) e in parte per l'esponente da dare alla base del sistema di numerazione (10), utilizzando, ad esempio, 6 cifre per la mantissa e due per l'esponente.

Il numero dell'esempio precedente (823.45), risulterebbe, in questa rappresentazione:

```
08234502
```

Le costanti

Anche le costanti hanno il loro tipo che si distingue in base alla maniera in cui vengono scritte. Le costanti intere sono semplici successioni di cifre decimali. Occasionalmente si può avere la necessità di utilizzare notazione ottale o esadecimale facendo precedere le cifre, rispettivamente da `0` e da `0X`.

```
25  765364  -34  034225  0XF5B4
```

[9] Nell'esempio si utilizza una rappresentazione con cifre decimali anziché binarie a puro scopo esemplificativo; ricordate che non ha alcun senso dal punto di vista della rappresentazione di macchina.

Le costanti *floating-point* possono essere indicate in due modi: utilizzando esplicitamente il punto decimale oppure utilizzando il carattere `E`, iniziale di *exponent*, che sta ad indicare l'utilizzo della notazione scientifica.

```
34.2  3.7564E5  -3.4  -5.45E-2
```

3.9.5 Conversione di tipo

In un linguaggio tipato staticamente (in cui il controllo dei tipi è effettuato perlopiù a tempo di compilazione) ci si aspetta che, in certi contesti, appaiano tipi specifici.

Nel costrutto

```
a := EXPRESSION
```

ci si aspetta che l'espressione sul lato destro abbia lo stesso tipo di `a`. Così come nell'espressione

```
a + b
```

si vorrebbe che gli operandi fossero o entrambi interi o entrambi reali.

Cambiare tipo

Supponiamo di imporre che il tipo che ci si aspetta e il tipo che viene fornito siano **esattamente** lo stesso tipo. Se il programmatore volesse effettuare un'operazione come:

```
x = 10 + 4.5
```

dovrebbe assicurarsi di prevedere una **conversione di tipo** esplicita che può comportare conseguenze diverse a seconda dei tipi tra i quali facciamo conversione: dal non fare sostanzialmente niente a run-time al prevedere una conversione effettiva a livello di rappresentazione in memoria, nel caso, ad esempio, di interi e numeri *floating-point*.

Casting Quando si parla di conversione esplicita di tipo si utilizza il termine ***casting***. In C il casting viene effettuato con uno speciale operatore unario prefisso:

```
( NOME DEL TIPO ) ESPRESSIONE
```

L'espressione:

```
float x;
x = (float)234;
```

si interpreta, dal punto di vista semantico, come una conversione esplicita della costante intera `234` in una costante reale di egual valore, e nell'assegnamento di questo valore alla variabile reale `x`.

Conversione implicita Buona parte dei linguaggi, e il C è tra questi, non richiedono, però, l'esatta equivalenza dei tipi per effettuare assegnamenti o per eseguire operazioni. È sufficiente che il tipo utilizzato sia **compatibile** con quello che dovrebbe essere presente nel contesto.

La definizione di compatibilità, parte integrante del sistema di tipi del linguaggio, è più o meno lasca a seconda del linguaggio, non stupisce che in C sia una delle più lasche in assoluto. Tornando all'esempio:

```
x = 10 + 4.5
```

i tipi intero e reale sono compatibili tra di loro e quindi la conversione di tipo viene effettuata implicitamente dall'ambiente effettuando quella che si chiama ***type coercion***.

Proprio a causa del suo debole sistema di controllo dei tipi, C effettua un numero molto elevato di conversioni implicite, che possono essere anche abbastanza incomprensibili se non si pensa alla rappresentazione numerica che ci sta dietro.

Vediamo alcuni esempi interessanti.

In assegnamenti come:

```
short sh;
unsigned long int lo = 967483635;
sh = lo;[a]
lo = sh;[b]
```

[a] i bit meno significativi di `lo` sono interpretati come numero con segno (`sh` vale `-24333`)

[b] `sh` è esteso, mantenendo il segno, alla lunghezza di `lo` e poi interpretato come numero senza segno (`lo` vale `4294942963`)

Ancora:

```
float fl;
double dou = 4.56783225526e+052;
unsigned long int lo = 1315696199;
fl = lo;[a]
fl = dou;[b]
```

[a] `lo` è convertito a *floating-point* ma poiché `fl` ha un minor numero di cifre significative ci può essere perdita di precisione (informazione) durante la conversione non controllabile dal programmatore (`fl` vale `1.31569626e+009`)

[b] `dou` è convertito ad un formato più corto (con meno bit), ci può essere perdita di precisione e, se il valore di `dou` non può essere rappresentato in singola precisione, il risultato è **indefinito** senza che si generi un errore a run-time (`fl` vale `inf`!)

3.9.6 Definire nuovi tipi

In molti linguaggi ci sono meccanismi espliciti che consentono di creare nuovi tipi a partire dai tipi base forniti all'interno del linguaggio. La creazione di nuovi tipi è alla base della definizione di equivalenza tra tipi.

In C il costrutto che consente di creare nuovi tipi è un particolare tipo di dichiarazione:

`typedef` $\text{TIPO}_{\text{existing}}$ TIPO_{new}

Questa dichiarazione ha come effetto che il compilatore aggiunga alla tavola dei tipi ammissibili all'interno del programma corrente anche il nuovo tipo appena dichiarato. Per convenzione è buona norma utilizzare, per i tipi dichiarati dall'utente, dei nomi con iniziale maiuscola.

Come usare i nuovi tipi

Vediamo perché e come si possono utilizzare tipi creati dall'utente con un esempio. Se vogliamo utilizzare, in un programma, variabili che consentono di manipolare quantità di denaro potrebbe essere utile avere i seguenti costrutti

```
#define 20Euro 20.0
#define 50Cent 0.50
...
typedef float Euro;
Euro bill = 20Euro;
Euro coin = 50Cent;
```

Il vantaggio rispetto all'usare il tipo `float` è puramente di forma, dato che nella sostanza abbiamo definito una completa equivalenza tra il tipo `float` ed il tipo `Euro` e possiamo utilizzarli con un'interscambiabilità completa:

```
float f;
f = bill + coin;
```

Enumerazioni

Un modo alternativo per definire un nuovo tipo, obbligatoriamente discreto, è costituito dalle **enumerazioni**, un costrutto introdotto per la prima volta nell'Algol W. Le enumerazioni facilitano la creazione di programmi leggibili e consentono al compilatore di rilevare errori di programmazione (*type clash*) altrimenti non identificabili.

Un tipo costruito come enumerazione consiste in un insieme di elementi con nome. In Pascal, ad esempio, per definire i nomi dei giorni della settimana, si scriverebbe:

```
type weekday =(sun, mon, tue, wed, thu, fri, sat);
```

Dopo questa dichiarazione `weekday` è un tipo del linguaggio come i tipi predefiniti e le regole del sistema di tipi si applicano anche ad esso, una variabile di tipo `weekday`, quindi, potrà assumere solo valori all'interno dell'insieme di enumerazione.

I valori all'interno dell'insieme definito con un'enumerazione sono **ordinati** ed è quindi possibile effettuare dei confronti che restituiscono un valore logico valido:

```
mon < tue
```

Dato che è possibile anche determinare successore e predecessore (è per questo che le enumerazioni sono tipi **discreti**) sono tipi facilmente utilizzabili per scrivere cicli controllati, e non è un caso, dall'enumerazione:

```
for today := mon to fri do
   begin
   ...
   end
```

Enumerazioni in C Anche C mette a disposizione una dichiarazione di enumerazione, che, purtroppo, non svolge un ruolo così potente come in Pascal. Un'enumerazione in C è, infatti, semanticamente equivalente alla dichiarazione di un insieme di costanti:

```
enum weekday {sun, mon, tue, wed, thu, fri, sat}
```

equivale a dichiarare

```
typedef int weekday;
const weekday sun=0,mon=1,tue=2,wed=3,thu=4,fri=5,sat=6;
```

Il vantaggio di usare il costrutto `enum` sta nella maggiore compattezza nella dichiarazione del set di valori che costituiscono l'enumerazione.

Enumerazioni come nuovi tipi Utilizzare le enumerazioni senza associarle alla creazione di nuovi tipi rende abbastanza farraginoso assegnare delle variabili destinate a contenere uno tra i valori dell'enumerazione. Una soluzione semplice ed efficace è quella di associare all'enumerazione un nuovo tipo:

```
typedef enum {sun,mon,tue,wed,thu,fri,sat} Weekday
```

Dopo aver definito questo nuovo tipo è possibile dichiarare delle variabili di tipo `Weekday` e, ovviamente, anche inizializzarle:

```
Weekday today = sun;
```

Nonostante tutto, `today` rimane una variabile intera con cui si possono compiere operazioni che non hanno senso logico, per come è stata dichiarata, quali:

```
k = i + today;
today = 23;
```

3.10 Vettori

Le variabili che abbiamo definito sino ad adesso sono caratterizzate dalla proprietà di avere associato, in ogni istante della computazione, uno ed un solo valore del tipo opportuno. È spesso utile, all'interno di una computazione, avere a disposizione n variabili, dello stesso tipo, che sono destinate ad essere associate a valori che hanno un qualche collegamento tra di loro. Pensiamo, ad esempio, alla possibilità di avere a disposizione tutti i voti conseguiti da uno studente, durante la sua carriera universitaria, all'interno di un programma per fare elaborazioni quali il calcolo della media, o ricavare il voto più frequente conseguito. La soluzione che potremmo ipotizzare è quella di avere tante variabili quanti sono i voti da memorizzare:

```
int voto1, voto2, voto3, ..., voton;
```

3.10.1 Vettori monodimensionali

Non è esattamente ciò che è più utile per effettuare calcoli in maniera semplice, si ha comunque a disposizione un simbolo diverso per ognuno dei valori utilizzati, sarebbe molto più utile avere un **unico simbolo** che possa essere utilizzato per fare riferimento a tutti i valori. La soluzione a questo problema è data dall'introduzione dei **vettori** (*array*), che sono collezioni di elementi, tutti dello stesso tipo, ognuno dei quali può essere indirizzato utilizzando il nome del vettore e un indice numerico che indica la posizione all'interno della collezione.

La sintassi che consente di **dichiarare** vettori in C è:

TIPO $\text{NOME}_{\text{vettore}}$ [$\text{DIMENSIONE}_{\text{vettore}}$]

dove TIPO è un tipo predefinito del linguaggio o dichiarato dall'utente, $\text{NOME}_{\text{vettore}}$ rispetta le stesse regole sintattiche dei nomi delle variabili e $\text{DIMENSIONE}_{\text{vettore}}$ è un'espressione intera costante.

Un esempio di dichiarazione di vettore è

```
int a[10];
```

che dichiara un vettore di interi di dimensione 10 chiamato `a`.

Gli elementi del vettore sono accessibili utilizzando l'**operatore di indicizzazione**, costituito da un'espressione intera racchiusa tra parentesi quadre:

```
a[i] = 23;
```

Ci sono due osservazioni importanti da fare:

1. Mentre l'espressione che appare nella **dichiarazione** del vettore **deve** essere costante perché viene valutata a tempo di compilazione, l'espressione utilizzata per **accedere** ad un elemento del vettore può essere qualsiasi, purché intera;
2. Il primo elemento di un vettore, in C, ha sempre indice `0`, non `1`.

Out of bound

Rispettando la filosofia di progettazione del linguaggio, C non effettua nessun controllo sul corretto uso degli indici. Sarebbe abbastanza semplice definire un controllo a run-time per evitare che vengano accedute degli elementi del vettore che non esistono, invece questo controllo è lasciato a carico del programmatore.

In un vettore di 10 posizioni, espressioni quali:

```
a[-1] = 0;
a[152] = 12;
```

sono sintatticamente corrette e non sollevano nessuna eccezione a run-time.

Il comportamento del programma in questo caso è completamente indefinito, dato che si vanno a modificare locazioni di memoria che non sono sotto il controllo nel nostro programma in esecuzione. Se si genera un errore di *segmentation fault* a run-time significa che si è acceduto a una locazione di memoria non utilizzabile dal programma e spesso una mancanza di controllo degli indici dei vettori può essere la fonte di questo errore.

Inizializzazione

Così come per le variabili, anche per i vettori è possibile (consigliabile!) assegnare un valore all'atto della dichiarazione. Per effettuare un'inizializzazione si devono racchiudere i valori iniziali (una lista di espressioni costanti) da assegnare ai singoli elementi del vettore tra parentesi graffe, separate da virgole:

```
int a[10]  = {23, 24, 22, 18, 27, 28, 30, 24, 24, 23};
```

Con il sistema di inizializzazione dei vettori si dimostra ancora una volta tutta la filosofia di mancanza di controllo che è stata alla base della progettazione del linguaggio C:

1. Se la lista contiene più elementi di inizializzazione degli elementi del vettore non c'è errore, solo un warning;

2. Se la lista contiene meno elementi degli elementi del vettore, gli altri vengono inizializzati a `0`;

3. Utilizzando una lista di inizializzazione si può omettere la dimensione del vettore che viene ricavata dalla scansione della lista.

Vediamo alcuni esempi di inizializzazioni sintatticamente corrette ma non conformi alla logica del comando.

```
int a[10] = {23,24,22,18,27,28,30,24,24,23,23,18,26};
```

La lista contiene più elementi di inizializzazione (13) rispetto alla dimensione del vettore (10) e il compilatore dà un warning[10]

```
int a[10]  = {23, 24, 22, 18, 27, 28}
```

La lista contiene meno elementi rispetto alla dimensione del vettore, per cui gli altri vengono inizializzati a `0`, senza alcun warning per il programmatore.

```
int a[] = {23, 24, 22, 18, 27, 28, 30, 24, 24, 23}
```

Se si omette la dimensione del vettore, la stessa viene ricavata verificando quanti elementi si trovano nelle lista di inizializzazione.

3.10.2 Vettori a più dimensioni

L'organizzazione delle informazioni utilizzando vettori può essere resa ancora più flessibile introducendo **vettori multidimensionali** detti anche **matrici**. La sintassi per dichiarare vettori multidimensionali in C è:

TIPO $\text{NOME}_{\text{vettore}}$ [DIM_1] [DIM_2] ... [DIM_n]

dove ogni DIM_i è il numero di elementi nella dimensione i.

Ad esempio

```
int fm[1024][1280];
```

dichiara una matrice con 1024 righe e 1280 colonne, per un totale di 1,310,720 elementi utilizzabili, ciascuno dei quali della dimensione di una variabile intera.

[10] Per gcc è `excess elements in array initializer`.

Accesso ad elementi di vettori multidimensionali

L'operatore di indicizzazione è un diretta estensione dell'operatore utilizzato per vettori ad una dimensione. Per ogni dimensione si deve avere un'espressione intera racchiusa tra parentesi quadre. Se vogliamo, quindi, accedere all'elemento che sta all'incrocio tra la i-esima riga e la j-esima colonna dovremo utilizzare la notazione:

```
a[i][j] = 564;
```

È da notare che dal punto di vista sintattico sarebbe forse stato più logico che si utilizzasse la notazione `a[i, j]`, più simile alla normale notazione per l'indicizzazione di matrici utilizzata in algebra lineare. Esiste una motivazione per utilizzare questa notazione, che vedremo in seguito quando parleremo di accesso diretto alla memoria.

Inizializzazione di vettori multidimensionali

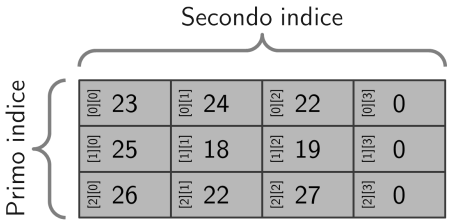
Nel caso dei vettori multidimensionali la lista di inizializzazione è ancora una volta formata da un numero di elementi pari al numero di elementi della matrice, e la sintassi con cui deve essere scritta prevede di separare ogni dimensione con un livello di parentesi graffe:

```
int a[3][4]  = {{23, 24, 22, 28},
                {25, 18, 19 ,22},
                {26, 22, 27 ,30}};
```

La mancanza di controllo sull'esattezza dell'inizializzazione, nel caso di vettori multidimensionali può generare degli errori che passano inosservati. Se la matrice precedente viene inizializzata invertendo l'ordine delle righe e delle colonne:

```
int a[3][4]  = {{23, 24, 22},
                {25, 18, 19},
                {26, 22, 27},
                {24, 25, 27}};
```

si avranno tutti gli ultimi valori di ogni riga inizializzati a `0` e l'ultima terna di dati di inizializzazione non utilizzata:



3.11 Subroutine

L'**astrazione** è il processo attraverso il quale il programmatore può associare un nome ad un frammento potenzialmente complicato del programma, che può quindi essere pensato in termini del suo scopo o della sua funzione, piuttosto che in termini della sua implementazione. Si può distinguere tra:

Astrazione di controllo, in cui lo scopo principale dell'astrazione è quello di eseguire un'operazione ben definita;

Astrazione dei dati, in cui lo scopo principale dell'astrazione è quello di rappresentare le informazioni.

Le **subroutine** sono il principale meccanismo per l'astrazione di controllo nella maggior parte dei linguaggi di programmazione. Una subroutine esegue il suo compito per conto del **chiamante** che attende la fine della subroutine prima di continuare nell'esecuzione. La maggior parte delle subroutine hanno **parametri**: il chiamante passa gli argomenti che influenzano il comportamento della subroutine, oppure fornisce in altro modo, ad esempio utilizzando memoria condivisa, i dati su cui operare. Il principio più rilevante dell'uso di subroutine (e di altri meccanismi di astrazione del controllo) è che non è importante sapere **come** la subroutine opera, ma **cosa** compie, con che dati e che risultati produce. È un meccanismo a cui spesso ci si riferisce come ***black box***.

3.11.1 Astrazione del controllo

Supponiamo che, nel corso della progettazione del nostro programma, ci si renda conto di avere la necessità di effettuare **ripetutamente** una particolare computazione, ad esempio il calcolo di quanti chilometri di autonomia ha ancora la nostra macchina basandosi sulla velocità corrente. Supponiamo anche che informazioni sulla capacità del serbatoio e sul suo contenuto corrente siano

disponibili alla porzione di codice che effettua la computazione senza avere la necessità di fornirli esplicitamente.

Dovremo costruire un'astrazione di controllo con l'ipotesi che, sulla base dell'indicazione della velocità in chilometri all'ora ci restituisca l'autonomia residua. L'aspetto interessante di un approccio come questo è dato dalla possibilità, a livello di progettazione iniziale, di poter evitare di definire nel dettaglio le modalità con cui il calcolo viene effettuato, ma solo di dichiarare che vogliamo che questo calcolo venga demandato ad una particolare subroutine In altre parole affido una parte del carico di computazione ad una porzione di programma che non ho ancora progettato basandomi sull'assunzione che comunque rispetti le specifiche richieste.

3.11.2 Argomenti delle subroutine

Quando si dichiara una subroutine si deve dichiarare anche la lista dei suoi parametri, detti **parametri formali** dichiarando per ciascuno di essi di quale tipo sia. Gli argomenti delle subroutine sono detti **parametri attuali** e sono fatti corrispondere ai parametri formali della subroutine al momento della sua invocazione. Una subroutine che restituisce un valore è detta **funzione**, una subroutine che non restituisce un valore è chiamata **procedura**.

La maggior parte dei linguaggi richiedono che le subroutine siano dichiarate prima di essere utilizzate, anche se alcuni (tra cui Fortran, C, e Lisp) non lo richiedono. Le dichiarazioni consentono al compilatore di verificare che ogni chiamata a una subroutine sia coerente con la sua dichiarazione, per esempio, che passi il giusto numero ed i tipi corretti di argomenti.

Dichiarazione della subroutine

La prima decisione che dobbiamo prendere sarà quindi legata non tanto alla modalità con cui implementeremo il calcolo ma ai suoi dati di input e di output. Decidere quali sono questi dati è quella che si chiama **dichiarazione** della subroutine.

Nel nostro caso potremo decidere che la velocità, dato che può essere arrotondata senza perdita evidente di precisione di calcolo, sia un numero intero, così come l'autonomia in chilometri rimasta. Vale la pena notare che, d'altronde, questa è la scelta standard effettuata dalla maggior parte dei veri sistemi di controllo delle automobili. Tradotto in termini implementativi, questo vorrà dire che la subroutine `autonomia` avrà in ingresso un numero intero e produrrà come risultato ancora un numero intero.

3.11.3 Subroutine in C

La sintassi della **dichiarazione** di una subroutine in C è

$$\text{TIPO}_{\text{ritorno}}\ \text{NOME}_{\text{subr}}\ (\ \text{TIPO}_{\text{param}_1}, \text{TIPO}_{\text{param}_2}, \ldots, \text{TIPO}_{\text{param}_n}\ )$$

dove $\text{TIPO}_{\text{ritorno}}$ è il tipo del valore che la subroutine restituisce e la lista tra parentesi tonde è la lista dei tipi che avrà ciascun parametro formale della subroutine.

Un tipo particolare di valore che può assumere $\text{TIPO}_{\text{ritorno}}$ è il tipo `void`. In effetti `void` non è un **vero** tipo, ma solo un'indicazione al compilatore che la subroutine non ritornerà alcun valore (nella nomenclatura che utilizziamo è una procedura) e quindi che non dovrà obbligatoriamente terminare con un'istruzione `return`.

Dichiarazione della subroutine

Proviamo quindi adesso a dichiarare la subroutine `autonomia` descritta in precedenza:

```
int autonomia(int);
```

Abbiamo detto che, dal punto di vista sintattico, la dichiarazione di una subroutine in C non è obbligatoria, il compilatore possiede dei meccanismi di salvaguardia che consentono di adoperare anche delle subroutine che non sono state dichiarate. Ma la regola migliore è quella di dichiarare **sempre** le subroutine prima di definirle ed utilizzarle. Inoltre, per maggiore chiarezza di costruzione del codice, è bene utilizzare un apposito file *header* (ad esempio `myprogram.h`) che contiene la dichiarazione di tutte le subroutine definite all'interno del corrispondente file `.c` (in questo caso `myprogram.c`).

Definizione della subroutine

Anche se il primo passo, essenziale, di progettazione, consiste nella decisione di quali siano i valori di input ed output della subroutine è vero comunque che non è possibile utilizzarla sino a quando non ne sia stato definito il contenuto. Decidere gli aspetti implementativi della subroutine, ovvero scrivere effettivamente il codice ad esso associato, è quella che si chiama la **definizione** della subroutine.

Definire una subroutine è un'operazione esattamente uguale a quella che abbiamo effettuato sino ad adesso per costruire i nostri programmi. In effetti il programma come l'abbiamo inteso sino ad ora può essere pensato come una subroutine di nome `main` invocata, senza parametri, dall'ambiente a run-time.

La sintassi della **definizione** di una subroutine in C è:

$$\text{TIPO}_{\text{ritorno}}\ \text{NOME}_{\text{subr}}\ (\ \text{TIPO}_{\text{param}_1}\ \text{NOME}_{\text{param}_1}, \ldots, \text{TIPO}_{\text{param}_n}\ \text{NOME}_{\text{param}_n}\ )$$

In definizione si riprende la lista dei parametri indicata in dichiarazione ma, mentre in dichiarazione è necessario solo conoscere qual è il **tipo** di ogni

parametro che viene utilizzato per l'invocazione della subroutine, in definizione ogni parametro deve avere, oltre al tipo, un **nome** che viene utilizzato all'interno dell'ambiente della subroutine.

Possiamo pensare ad ogni **parametro formale** come a un tipo particolare di variabile che viene inizializzata con il valore che si utilizza per invocare la subroutine. In effetti, all'interno della subroutine, il parametro formale è esattamente equivalente a una variabile dichiarata a inizio subroutine.

Quando definiamo la subroutine `autonomia`, quindi, dobbiamo dare un nome all'unico parametro formale utilizzato e poi definire la subroutine:

```
int autonomia(int kmh)
{
   int auton;
       eseguo tutti i calcoli necessari per ottenere
       il valore di auton usando kmh
   return auton;
}
```

Dato che la subroutine è una funzione, ovvero ritorna un valore, l'ultima istruzione che esegue deve essere un'istruzione `return` seguita da un'espressione che, valutata, dà un valore intero, in questo caso la variabile `auton`.

Prima definire e poi utilizzare?

La dichiarazione della subroutine deve essere precedente a qualsiasi suo utilizzo, la sua definizione, invece, non necessariamente. Se pensiamo infatti ad un caso, non così raro, di due subroutine che si invocano vicendevolmente, si capisce come non sia possibile imporre che una subroutine venga definita prima di essere utilizzata. In effetti, ragionandoci su, non sarebbe affatto necessario imporlo. Quello che è veramente necessario, per l'utilizzatore di una particolare subroutine, è definire correttamente la lista dei parametri che vengono utilizzati all'invocazione. La dichiarazione è essenziale affinché il compilatore definisca **come** vengono collegati chiamante e chiamato, il collegamento effettivo lo realizza il linker una volta che tutte le porzioni di codice sono state compilate.

Utilizzo della subroutine

La modalità con cui viene utilizzata una subroutine sono semplici. Una volta dichiarato, il nome della subroutine entra a far parte della lista dei nomi riconosciuti dal compilatore[11], come i nomi delle variabili o dei nuovi tipi dichiarati dall'utente. Questo nome, dal punto di vista semantico, è paragonabile ad un nuovo **operatore prefisso** che, una volta applicato ad una lista di argomenti elencati tra parentesi dopo il nome, restituisce un valore del tipo indicato nella

[11] A questo scopo ricordiamo ancora una volta che viene utilizzata la tavola dei simboli.

dichiarazione. Anche in questo caso, come sempre in C, sono possibili **conversioni implicite** di tipo nel caso in cui gli argomenti non siano di un tipo equivalente al tipo indicato nella lista di dichiarazione.

Una porzione di codice che invoca l'esecuzione della subroutine `autonomia` potrebbe essere la seguente:

```
int i;
i = autonomia(50*2);
```

La sua interpretazione è la seguente:

1. Si valuta l'espressione intera `50*2` che ritorna il valore `100`;
2. La subroutine `autonomia` viene invocata assegnando al parametro formale `kmh` il valore `100`;
3. La subroutine viene eseguita e, nel frattempo, il programma chiamante interrompe l'esecuzione in attesa del valore di ritorno;
4. Si assegna il valore di ritorno alla variabile `i`.

3.11.4 Passaggio di parametri

Supponiamo di voler effettuare una chiamata della subroutine `p` con il parametro `x`. Ci sono due modi fondamentali per mettere a disposizione il valore di `x` a `p`:

1. Fornire a `p` il valore che ha `x` nel momento in cui si invoca `p`, questo si chiama passaggio di parametri per **valore**;
2. Fornire a `p` un riferimento alla posizione in memoria che ha `x` nel momento in cui si invoca `p`, questo si chiama passaggio di parametri per **riferimento**.

Con i parametri passati per **valore**, ogni parametro attuale è assegnato al corrispondente parametro formale quando una subroutine viene chiamata, da quel momento in poi i due sono indipendenti. Con i parametri passati per **riferimento**, ogni parametro formale introduce, all'interno del corpo della subroutine, un nuovo nome per il corrispondente parametro attuale.

In C esiste una sola modalità di passaggio di parametri, il passaggio per **valore**. Il meccanismo di passaggio di parametri è sufficientemente semplice, le seguenti operazioni vengono eseguite per ciascun parametro della lista:

1. Si valuta l'espressione che costituisce il parametro attuale della subroutine;
2. L'ambiente a run-time esegue un assegnamento della variabile parametro formale della subroutine con il valore dell'espressione valutata al punto precedente;

3. Si utilizza per la computazione, all'interno della subroutine, il parametro formale;

4. Al termine dell'esecuzione della subroutine, nel momento in cui il controllo passa di nuovo al chiamante, la variabile parametro formale cessa di avere un valore utilizzabile.

Vediamo con un esempio come si esplica il meccanismo del passaggio di parametri per valore:

```
int main()
{
   int i = 10, m;[a]
   m = foo(i);[b]
   return 0;[c]
}

int foo(int k)
{
   k = k*10;[d]
   return k;[e]
}
```

[a] si dichiarano `i` e `m`, **visibili** solo dentro main
[b] si invoca `foo` con argomento `i`; `i` **non è stato modificato** dall'invocazione di `foo`, `m` invece sì
[c] terminata l'esecuzione `main` ritorna `0`
[d] `k` viene moltiplicato per `10` e riassegnato a `k`
[e] terminata l'esecuzione `foo` ritorna `k` che ora vale `100`

Passaggio parametri come assegnamento Ciò che avviene al momento dell'invocazione di una subroutine con parametri è quindi una combinazione di due azioni:

1. Si effettua una serie di assegnamenti di variabili utilizzando come **lvalue** la lista dei parametri formali e come **rvalue** la lista degli argomenti (parametri attuali);

2. Si trasferisce il controllo dall'istruzione in cui avviene l'invocazione della subroutine alla prima istruzione eseguibile della subroutine stessa.

I meccanismi con cui il compilatore e l'ambiente a run-time gestiscono la **visibilità dei simboli** ed il **passaggio di controllo** sono definiti dalle regole di **scope** e dalla modalità con cui viene salvato il **contesto di esecuzione** quando si invoca una subroutine.

3.12 Scope

Lo *scope*[12] di un identificatore è lo spazio, all'interno del programma, in cui è definito e, quindi, utilizzabile.

3.12.1 Nomi

Un **nome** non è altro che una stringa di caratteri che viene utilizzata come ausilio mnemonico per rappresentare oggetti di varia natura all'interno del proprio ambiente di programmazione. In molti linguaggi i nomi sono **identificatori** (token alfanumerici), anche se alcuni altri simboli, come ad esempio + o := possono essere nomi. I nomi ci consentono di fare riferimento a variabili, costanti, operazioni, tipi, e così via con identificatori simbolici piuttosto che usando strumento di più basso livello come gli indirizzi, che sono strettamente legati all'architettura della macchina su cui si opera. Sono il primo e più importante meccanismo di astrazione che permette il passaggio da linguaggi assemblativi a linguaggi ad alto livello.

3.12.2 Binding

Il processo attraverso il quale si effettua l'associazione tra un nome e l'oggetto che ha quel nome è il ***binding***.

I momenti in cui si può effettuare il binding sono vari e vanno dalla progettazione del linguaggio (in cui si decide, ad esempio, quali sono i costrutti e la semantica del linguaggio stesso e anche quali nomi riservare al linguaggio) al momento in cui si esegue un programma scritto in quel linguaggio (a run-time si effettuano le assegnazioni di valori a variabili nonché una miriade di altre operazioni).

Quello che ci interessa adesso è vedere il meccanismo che consente di manipolare gli oggetti a cui viene assegnato un nome.

Modalità di binding

Le due modalità di binding che ci interessa analizzare sono: il binding **statico**; il binding basato su uno ***stack*** (pila).

Oggetti a cui viene assegnato un nome in maniera statica sono associati ad un indirizzo assoluto (all'interno del blocco dati del programma) in memoria che viene mantenuto per tutta l'esecuzione del programma e sono **visibili in qualsiasi momento**.

Oggetti a cui viene assegnato un nome utilizzando una pila sono associati ad indirizzi di memoria che possono variare a seconda delle circostanza e **non sono sempre necessariamente visibili**.

[12] Si usa quasi sempre la parola inglese, raramente la traduzione italiana ambito.

3.12.3 Più oggetti con lo stesso nome

Supponiamo di aver iniziato a progettare il nostro programma in maniera modulare ed aver delegato l'implementazione di porzioni di programma (subroutine) ad altri programmatori.

Supponiamo inoltre che, nel nostro programma, stiamo utilizzando anche funzioni o procedure di libreria, scritte quindi da chi ha messo a disposizione l'ambiente di sviluppo.

Come facciamo ad essere sicuri che non ci sia la possibilità che un certo nome, che ho deciso di utilizzare nel mio programma, non sia stato già utilizzato all'interno di una subroutine di libreria o non venga anche utilizzato dagli altri programmatori che scrivono le subroutine?

La soluzione a questo problema consiste nel definire delle **regole di visibilità** dei nomi che garantiscono che conflitti di questo tipo non sorgano. La regione (testuale) di programma in cui il binding di un nome con un oggetto è attiva è lo ***scope*** di quell'oggetto.

Vediamo un esempio di scope in C (i box indicano lo scope dei nomi):

```
int main()
{
    int a;[a]
    ...
}
int foo(int k)
{
    int a;[b]
    ...
}
```

[a] `a` è visibile all'interno dello scope di `main` ma non di `foo`

[b] questa `a` è visibile all'interno dello scope di `foo` ma è **un altro oggetto con lo stesso nome** rispetto alla `a` dichiarata nel `main`

Qui lo stesso nome (`a`) si riferisce a due locazioni di memoria distinte.

Visibilità di blocco

Passando dallo spazio dei nomi di una subroutine[13] allo spazio dei nomi di un'altra si cambia completamente il contesto e tutti i nomi visibili. Sarà chiaro il perché dopo la lettura del contenuto della sezione 3.13.2.

Mentre in altri linguaggi come Pascal, come abbiamo visto, è obbligatorio dichiarare tutte le variabili all'inizio della subroutine, in C è possibile **aggiungere**, all'interno di una subroutine, dei nomi che sono visibili in uno spazio di

[13] La speciale subroutine che prende il nome di `main` in questo contesto è da considerare una subroutine come tutte le altre!

nomi ancora più ristretto dichiarandoli all'interno di un blocco. La sintassi del C99 consente addirittura di mescolare dichiarazioni e istruzioni all'interno del programma, creando delle situazioni in cui la possibilità di manutenzione del codice diminuisce notevolmente.

Un esempio di questo utilizzo:

```
int main()
{
   int a;[a]
   ...
   {
      int b;[b]
   ...
   }
}
```

[a] `a` è visibile all'interno dello scope `main`

[b] `b` è visibile solo all'interno dello scope del blocco in cui è dichiarata, in questo blocco è comunque visibile anche `a`

Questa possibilità è ancora una volta un retaggio dell'epoca in cui anche il solo risparmiare sull'allocazione di una o più variabili poteva dare un rilevante vantaggio di efficienza del codice, data la scarsità di memoria.

Mascheramento dei nomi

Una conseguenza della possibilità di aggiungere nuovi nomi all'interno dello spazio di blocco è il cosiddetto effetto di **mascheramento**:

```
int main()
{
   int a;[a]
   ...
   {
      int a;[b]
   ...
   }
}
```

[a] `a` è visibile all'interno dello scope `main`

[b] questa `a` è visibile solo all'interno dello scope del blocco in cui è dichiarata, in questo blocco **non** è visibile la `a` dichiarata all'esterno

Il riutilizzo dei medesimi nomi all'interno di spazi di nomi diversi è la norma della programmazione. La dichiarazione di variabili locali a un blocco è invece totalmente **sconsigliabile** e il mascheramento da **evitare assolutamente**.

3.12.4 Variabili globali

Tutti i nomi utilizzati sinora richiedono l'uso di uno stack per la gestione della visibilità (vedremo in seguito come). In C l'esempio più importante di oggetto statico sono le **variabili globali**.

Sinora abbiamo ipotizzato che la dichiarazione di una variabile potesse essere effettuata solo all'interno di un programma o un sottoprogramma, in effetti è possibile dichiarare delle variabili anche al loro esterno, sono le variabili globali.

Lo scope delle variabili globali è l'insieme di tutti gli scope definiti all'interno della porzione di testo a cui appartiene, ovvero il file in cui la dichiarazione è contenuta. I simboli sono visibili in tutto il codice che si incontra **dopo** la dichiarazione della variabile globale, in altre parole si tratta di un **scope di file**.

In linguaggi con una modularità diversa (es. object-oriented, come Java) non esiste lo scope di file ma solo di classe.

Un esempio di utilizzo di variabili globali con indicazione del loro scope è il seguente:

```
int a = 10;
int main()
{[a]
    a = 20;
    foo();
}
int foo(int k)
{
    ...[b]
}
```

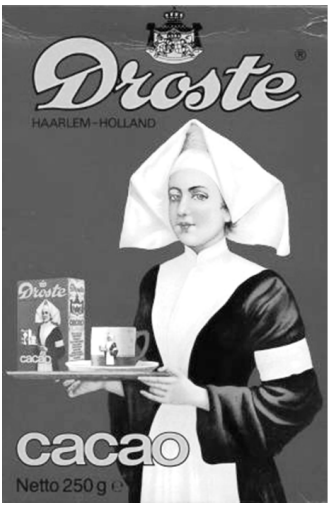
[a] Qui `a` è visibile e vale `10`

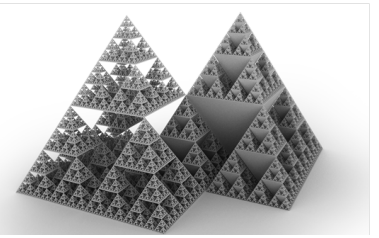
[b] Anche qui `a` è visibile e vale `20` se valutato subito dopo l'invocazione di `foo` effettuata nel `main`, altrimenti non sappiamo quale sia il suo valore

L'uso di variabili globali è una scorciatoia rispetto al passaggio di parametri alle subroutine e deve essere **il più possibile limitato**, dato che, formalmente, non è mai necessario e impedisce di effettuare una valutazione delle modalità di esecuzione di ogni subroutine in maniera indipendente l'una dall'altra.

3.13 Ricorsione

Il meccanismo che, in matematica e informatica, si chiama **ricorsione**, è un metodo di definizione delle funzioni, in cui la funzione che viene definita è applicata nella sua stessa definizione. Il termine può essere usato anche, più in generale, per descrivere il processo di ripetere gli oggetti in modo auto-simile:





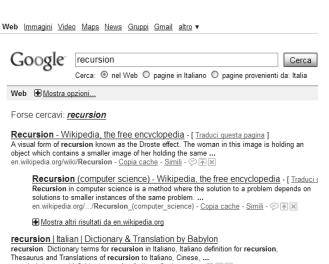
Il concetto di definizione ricorsiva può essere ritrovato, per fini seri o per umorismo, in altri campi della vita. Pensiamo, ad esempio, alla seguente definizione di procedimento per spostare una pila di n cassette di frutta da un posto A ad un altro B

1. Sposta la cassetta più in alto nella pila da A a B;
2. Controlla se ci sono altre cassette;
3. Se ci sono applica il procedimento che si sta definendo con le restanti cassette $(n-1)$, altrimenti lo spostamento è terminato.

Ad ogni passo il problema da risolvere ha **dimensione sempre più piccola** sin quando non si arriva al problema di **dimensione minima**. In questo caso quando non ci saranno più cassette da spostare.

Un divertente esempio di definizione ricorsiva si trova utilizzando Google per cercare il termine *recursion*

3.13.1 Computazione ricorsiva



Il campo in cui si utilizza più compiutamente il concetto di ricorsione è naturalmente la matematica. È abbastanza naturale, infatti, utilizzare la computazione ricorsiva legandola all'insieme dei numeri naturali, visto che, in $\mathbb{N}$, sono sempre definiti successore e predecessore e quindi si riesce facilmente a definire l'ordine della computazione.

Vediamo due esempi di computazione ricorsiva di funzioni matematiche, all'apparenza molto

simili tra loro ma che, una volta tradotti nei corrispondenti programmi che le calcolano, si rivelano alquanto diversi per la complessità delle operazioni e la possibilità di trasformare la ricorsione in un altro modello di computazione.

Calcolo del fattoriale

Il fattoriale $n!$ di un numero n positivo è definito come **il prodotto di tutti i numeri interi minori o uguali di** n. Per esempio:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Il calcolo del fattoriale si presta benissimo ad essere impostato come calcolo di una funzione ricorsiva:

$$n! = n \times (n-1)!$$

Un problema da risolvere è quello di determinare quando si deve interrompere la catena di invocazioni della funzione. In questo caso è abbastanza semplice convincersi che si interrompe quando si arriva all'ultimo numero della successione e definire che il fattoriale di 1 è 1: $1! = 1$.

Programma per il calcolo del fattoriale Tradurre questa computazione in un programma che effettua lo stesso calcolo non è molto complesso, si tratterà in primo luogo di dichiarare la funzione che calcola il fattoriale di un numero e non ci sono dubbi che sia una funzione che, dato un intero, restituisce un intero, appunto il suo fattoriale:

```
int fact(int)
```

Per la definizione si deve tradurre la formulazione matematica

$$n! = \begin{cases} 1 & \text{se } n = 1, \text{ caso base} \\ n \times (n-1)! & \text{se } n > 1, \text{ passo ricorsivo} \end{cases}$$

in linguaggio di programmazione:

```
int fact(int i) {
   int res;
   if (i>1)
      res = i*fact(i-1);[a]
   else
      res = 1;[b]
   return res;
}
```

[a] passo di ricorsione: si applica in tutti gli altri casi

[b] passo base: si applica nel caso in cui si verifica la prima condizione della formula

Analizziamo adesso le varie parti della subroutine che esegue il calcolo del fattoriale. La parte più rilevante è legata alla decisione di eseguire un passo di ricorsione oppure di eseguire la computazione di base: si deve verificare se l'argomento di cui si calcola il fattoriale è `1` o maggiore di `1`. Nel caso in cui si debba calcolare il fattoriale di `1` si restituisce `1` stesso. Nel caso, invece, in cui si debba calcolare il valore di un intero `i` maggiore di `1` si restituisce il prodotto di `i` per il fattoriale di `i-1`.

Ad ogni passo della computazione stiamo riducendo le dimensioni del problema che, alla fine, si ridurrà alle sue dimensioni minime, cioè a `1`.

In casi come questo del calcolo del fattoriale, in cui l'invocazione della funzione ricorsiva avviene una sola volta all'interno del corpo della funzione, possiamo dire come ultima azione, in coda alla funzione, la funzione si dice ***tail recursive***. Questo tipo di funzioni è facilmente trasformabile in un costrutto che non usa la ricorsione, ovvero in un costrutto iterativo:

```
int fact(int i) {
   int res = 1;
   if (i==1)
      res = 1;[a]
   else
      while (i>1) {[b]
         res = res * i;
         i = i-1;
      }
   return res;
}
```

[a] non sono richiesti calcoli, il risultato è 1

[b] calcolo iterativo che viene effettuato con `i` decrescente

La sequenza di Fibonacci

Un altro problema che si presta molto bene ad essere risolto mediante un calcolo ricorsivo è il calcolo del termine n-esimo della sequenza di Fibonacci. La sequenza dei numeri di Fibonacci parte con i numeri **0** e **1** e ogni numero seguente è dato dalla somma dei due precedenti:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 \ldots$$

Per descrivere la sequenza di Fibonacci è molto semplice ricorrere alla sua definizione ricorsiva:

$$F(n) = \begin{cases} 1 & \text{se } n = 0 \wedge n = 1 \\ F(n-1) + F(n-2) & \text{se } n \geq 2 \end{cases}$$

Programma per il calcolo della sequenza di Fibonacci Anche nel caso del calcolo dei numeri della sequenza di Fibonacci non è molto difficile scrivere il codice che effettua il calcolo dell'n-esimo elemento in maniera ricorsiva, la dichiarazione sarà:

```
int fib(int);
```

Ancora una volta, per la definizione si deve solo tradurre la formulazione matematica in linguaggio di programmazione:

```
int fib(int i) {
   int res;
   if (i>1)
      res = fib(i-1)+fib(i-2);[a]
   else if (i == 1)
           res = 1;[b]
        else
           res = 0;[c]
   return res;
}
```

[a] passo di ricorsione
[b] passo base per `1`
[c] passo base per `0`

La differenza sostanziale tra il calcolo del fattoriale e quello dei numeri di Fibonacci è data dal numero di invocazioni ricorsive della funzione che si effettuano all'interno del corpo: **una** nel caso del fattoriale, **due** nel caso di Fibonacci, La sostituzione di una singola chiamata di funzione per costruire un programma che utilizza iterazione è un'operazione semplice da effettuare, sia da parte del programmatore ma anche da parte del compilatore ottimizzante. La sostituzione di due o più chiamate per eliminare la ricorsione, invece, è molto meno banale. Comporta una riprogettazione della soluzione, calcolando la sequenza a partire dai suoi primi termini fino ad arrivare al termine cercato.

3.13.2 Call stack

Abbiamo parlato delle due modalità di binding **statico** e basato su **stack**. Mentre, come abbiamo visto, il binding statico è semplice da realizzare e ha come risultato che il nome (simbolo) definito staticamente è visibile in tutto lo scope del file in cui è dichiarato, il binding basato su stack è ben più complesso ma ci spiega il funzionamento del cambio di contesto in occasione di una invocazione di subroutine e, soprattutto, la possibilità di utilizzare la computazione ricorsiva.

Cos'è il contesto di esecuzione

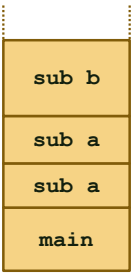
Il contesto di esecuzione di un programma è costituito da un insieme di informazioni che in sintesi si possono riassumere come:

1. L'**indirizzo di ritorno** a cui restituire il controllo dopo aver terminato la computazione all'interno del contesto attuale;
2. L'insieme delle **variabili locali** al contesto in cui si sta operando;
3. La lista dei **parametri**, se presenti, con cui la subroutine a cui il contesto si riferisce è stata invocata;
4. Altre informazioni di vario genere.

Lo spazio per questo insieme di informazioni deve essere riservato in memoria per ogni nuovo contesto che viene attivato. Al tempo stesso si deve fornire un meccanismo che consenta di mantenere il contesto corrente della porzione di programma chiamante. La soluzione per questi problemi è l'uso di uno stack.

Il call stack

Il contesto di esecuzione di una particolare subroutine è costituito solo dai simboli definiti nella subroutine stessa e, in un linguaggio che lo permette, le subroutine possono essere definite ricorsivamente. Una struttura che permette di supportare questo modello è uno stack in cui si accumulano i contesti.



Per ogni subroutine, compreso il programma principale, viene allocato un ***frame*** sullo stack che contiene l'insieme di informazioni necessarie per la sua esecuzione. Quando si crea un nuovo frame sullo stack, i precedenti non sono visibili. Al termine dell'esecuzione della subroutine il frame viene eliminato e si ritorna, tramite l'indirizzo memorizzato, al chiamante, ritornando un valore nel caso di una funzione.

Ricorsione e stack

L'uso dello stack per supportare la ricorsione è indispensabile. Senza lo stack non si potrebbe avere la possibilità di interrompere l'esecuzione di una funzione, invocare un'altra istanza della stessa funzione e sfruttare il risultato calcolato per poter continuare l'esecuzione della funzione che si era interrotta, tutto questo per un numero di volte che è limitato solo dalla dimensione dello stack stesso.

La limitazione nell'uso dello stack (e quindi della computazione ricorsiva) è data dalla quantità di memoria necessaria: ad esempio, per ogni invocazione ricorsiva della funzione `fact` si deve allocare un nuovo frame sullo stack, mentre per la corrispondente versione iterativa si ha necessità solo di allocare memoria per `una` variabile.

L'errore che, a run-time, indica che lo spazio per lo stack è terminato è indicato con il messaggio `stack overflow`.

3.14 Gestione della memoria

L'uso che abbiamo fatto sino ad adesso della memoria per la computazione è stato totalmente implicito e statico. Nel momento in cui dichiariamo una variabile diamo per scontato che si attivi un meccanismo tramite il quale:

1. Abbiamo a disposizione memoria sufficiente per contenere i valori del tipo a cui la variabile è associata;
2. L'abbiamo a disposizione quando serve e, probabilmente, **solo** quando serve, utilizzando i frame allocati alle subroutine che vengono rimossi quando la subroutine non è più in esecuzione.

Un meccanismo alternativo per avere disponibilità di memoria per allocare i dati necessari alla computazione prevede l'utilizzo di due nuovi strumenti:

1. Riferimenti diretti a locazioni di memoria o **puntatori**;
2. Allocazione dinamica di memoria.

3.14.1 Sezioni di memoria a disposizione del programma

Prima però è opportuno ricapitolare come viene gestita la memoria durante l'esecuzione di un programma. Possiamo dividere la memoria a disposizione del programma in tre parti principali:

Program section La sezione destinata a contenere il **programma** stesso, dove sono memorizzate tutte le istruzioni del programma, in questa sezione possiamo ipotizzare che siano memorizzati anche i dati associati a simboli statici (costanti, variabili globali);

Stack La sezione che permette di allocare e disallocare i frame che si utilizzano per l'invocazione delle subroutine;

Heap Letteralmente sarebbe il mucchio, è la sezione che mette a disposizione memoria al programma in blocchi di dimensione arbitraria.

Lo heap

A seconda del linguaggio di programmazione utilizzato, i metodi per avere a disposizione la memoria heap variano: in un linguaggio object-oriented, per esempio, istanziare un nuovo oggetto prevede che la memoria per esso utilizzata sia una porzione di heap.

La memoria heap può essere allocata sia in maniera **implicita** (**automatica**), come nel caso precedente, in cui il programmatore non ha il controllo effettivo di quanta memoria deve essere utilizzata e come, oppure in maniera **esplicita**, direttamente utilizzando costrutti del linguaggio per ottenere porzioni di memoria.

Nello stesso modo la memoria può essere automaticamente o esplicitamente **rilasciata** (o deallocata) ovvero rimessa a disposizione dello heap.

Ambiente vs programmatore Uno dei principali obiettivi che guidano chi vuole progettare un linguaggio di programmazione, è il decidere quante operazioni devono essere a carico del sistema nel suo complesso e quante a carico del programmatore. Nel corso dello sviluppo della teoria e della tecnica dei linguaggi si è cercato di spostare sempre più operazioni a carico del sistema per un motivo molto evidente: evitare il più possibile errori dovuti a distrazione, dimenticanza o, semplicemente, all'incapacità di gestire le situazioni.

Nella gestione della memoria questa possibile ambivalenza di scelta si rivela in maniera evidente: se la gestione della memoria heap è totalmente a carico del sistema la possibilità di errori si riduce a zero, se è totalmente a carico del programmatore si ottiene la massima efficienza possibile dato che, a run-time, non deve essere effettuato alcun controllo accessorio.

3.14.2 Allocazione dinamica della memoria e garbage collection

L'allocazione della memoria ad un programma può venire effettuata:

1. In maniera **statica**, come abbiamo già visto;
2. Tramite stack, obbligatoriamente in maniera automatica da parte del sistema, con blocchi di dimensione precalcolata a tempo di compilazione (i frame o record di attivazione);
3. In maniera **dinamica** riservando e rilasciando porzioni dell'heap in modo automatico o manuale.

Nel primo caso la memoria è destinata all'uso del programma per tutta la durata dell'esecuzione. Nel secondo si alloca un frame sullo stack quando si invoca la subroutine relativa e si rilascia quando la subroutine ha terminato la sua esecuzione. Nell'ultimo caso il momento in cui si alloca e rilascia la memoria è arbitrario, così come la dimensione dei blocchi allocati.

Garbage collection

Così come l'allocazione, anche il rilascio della memoria può essere gestito in maniera automatica o meno. Lasciare al programmatore il rilascio esplicito delle memoria una volta che ne ha terminato l'utilizzo ha come vantaggio la **semplicità** di realizzazione dell'ambiente a run-time, in cui non si deve programmare nessuna politica di gestione della memoria, e l'**efficienza** (velocità in esecuzione), dato che non è necessario effettuare alcun controllo sull'utilizzo o meno dei blocchi di memoria allocati dallo heap.

L'argomento a favore della deallocazione automatica della memoria (***garbage collection***) è principalmente uno: errori nella deallocazione manuale sono tra i bug più comuni e costosi in programmi del *mondo reale*.

La garbage collection è una caratteristica comune dei linguaggi funzionali e di scripting ma anche di linguaggi imperativi moderni come Java e C#. C, purtroppo, non dispone di meccanismi di garbage collection

3.15 Puntatori

Quale che sia la porzione di memoria che dobbiamo utilizzare (una porzione di memoria statica, di stack o di heap), esiste sempre la necessità di disporre di un qualche meccanismo di indirizzamento che consenta di accedervi.

Il meccanismo più semplice di indirizzamento è l'utilizzo di un **nome** (simbolo) come etichetta di una locazione di memoria di dimensione adeguata per contenere i dati associati a quel nome: è il concetto standard di variabile.

Il meccanismo alternativo di accesso alla memoria prevede di utilizzare non un'etichetta associata ad un contenitore ma di mantenere un riferimento (un indirizzo) in memoria a disposizione in una locazione con nome. Per accedere alla memoria anziché utilizzare direttamente il simbolo si usa il simbolo per ottenere l'indirizzo: è un **indirizzamento indiretto**.

Indirizzamento indiretto Introducendo l'indirizzamento indiretto si hanno due categorie di simboli che consentono di accedere a locazioni di memoria:

Le variabili che permettono, tramite il nome, di accedere direttamente al contenuto, a cui è associato un tipo che è stato stabilito al momento della dichiarazione dell variabile.

I puntatori che sono contenitori di tipo standard (*address*) e contengono degli indirizzi in memoria in cui potrò memorizzare e recuperare informazioni il cui tipo dovrà essere specificato esplicitamente con qualche meccanismo proprio del linguaggio.

Quando sorge la necessità di utilizzare i puntatori? Nel modello object-oriented, per esempio, è indispensabile utilizzare esplicitamente indirizzi di

memoria mutevoli, dato che gli oggetti sono allocati dinamicamente dallo heap e ad essi si fa riferimento tramite l'indirizzo che l'ambiente restituisce al momento dell'allocazione. Tutto questo risulta, comunque, trasparente al programmatore che non ha l'effettiva necessità di rendersi conto che il simbolo che manipola è legato ad un indirizzamento indiretto.

In un linguaggio imperativo/procedurale come il C l'utilizzo dei puntatori è legato, storicamente, alla possibilità di accedere esplicitamente a locazioni di memoria, tipico della programmazione in assembler. In questo modo si riesce a gestire blocchi di memoria allocati dinamicamente.

3.15.1 I puntatori in C

Il linguaggio C mette a disposizione i puntatori introducendo due operatori, entrambi unari e prefissi, che consentono di manipolare esplicitamente gli indirizzi.

Il primo operatore, l'operatore *star* `*`, è un operatore di **indirezione** che consente di ottenere, dall'indirizzo, l'oggetto a cui l'indirizzo fa riferimento.

Il secondo operatore, l'operatore *ampersand* `&`, è l'operatore **indirizzo** che permette di effettuare l'operazione inversa: dato un oggetto ne restituisce l'indirizzo in memoria

Per avere a disposizione simboli di tipo puntatore si usa una sintassi particolare nella dichiarazione della variabile, facendo precedere il nome da un carattere `*`, o, meglio, facendo seguire il carattere `*` al tipo:

```
int *p; int* q;
```

Variabili dichiarate con questa sintassi **non** sono variabili intere, ma variabili che **contengono l'indirizzo** di locazioni di memoria che contengono interi.

Uso dei puntatori

Anche se non ha granché senso, è opportuno iniziare a prendere dimestichezza con la manipolazione diretta degli indirizzi senza utilizzare l'allocazione dinamica della memoria, per motivi di semplicità.

L'esempio standard che si può fare è la possibilità di utilizzare un puntatore per fare esplicito riferimento alla locazione di memoria che è stata riservata ad una variabile.

```
int i, *p;[a]
...
i = 10;[b]
p = &i;[c]
```

[a] due simboli con semantica diversa
[b] assegno un valore alla variabile `i`
[c] adesso `p` **punta** ad `i`

In questo modo abbiamo stabilito un'equivalenza tra `i` e `*p`, ovvero la locazione che si ottiene per indirezione utilizzando il simbolo `p`: operare sull'uno o sull'altro è indifferente.

```
i = 20;[a]
*p = 30;[b]
```

[a] modifico il valore della variabile `i`
[b] ma anche così modifico il valore di `i`, **non** di `p`!

Alias

Effettuare un'operazione come quella descritta si dice che crea un **alias** della variabile. Il numero di alias che possiamo utilizzare è qualunque, possiamo utilizzare, ad esempio, due o più alias per la stessa variabile:

```
int i, *p, *q;
...
p = &i;[a]
q = p;[b]
```

[a] `p` **punta** a `i`
[b] anche `q` **punta** a `i`

A questo punto gli alias di `i` sono due: `*p` e `*q`.

È da notare la differenza che c'è tra `q = p` e `*q = *p`: nel primo caso si creano due alias dello stesso oggetto, nel secondo caso si effettua un'operazione di assegnamento tra due variabili in maniera alquanto bizzarra.

Puntatori come argomenti

Sappiamo che in C il meccanismo di passaggio di parametri è solo per **valore**, non si può quindi utilizzare il passaggio per **riferimento** che comporta la condivisione di un medesimo riferimento alla memoria tra il chiamante e il chiamato.

Il passaggio per valore determina invece che, quando si invoca una subroutine, si effettua un cambio di contesto, creando un nuovo frame, avendo poi a disposizione, nel nuovo contesto, un simbolo (il parametro formale) associato inizialmente al medesimo valore associato, nel frame chiamante, al parametro attuale. Che succede se questi due parametri sono due puntatori?

I due parametri, formale ed attuale, diventano alias della medesima locazione di memoria e quindi nel frame del chiamato si ha a disposizione lo stesso riferimento alla memoria che sia aveva nel frame del chiamante.

Si può quindi dire che il linguaggio C possiede un meccanismo di passaggio di parametri per riferimento? **No!** Anche se per il programmatore il risultato può apparire il medesimo la differenza è sostanziale.

Il fatto che il parametro attuale, nel frame del chiamato, sia un valore, fa sì che possa essere **modificato**, perdendo, in questo modo, il collegamento creato, in maniera fittizia, utilizzando lo stesso riferimento alla medesima locazione di memoria.

Un'ulteriore conseguenza di questo meccanismo è la possibilità di modificare frame *frozen* sullo stack utilizzando riferimenti a locazioni di memoria in essi contenute. Non è altro che uno dei tanti meccanismi *quick&dirty* utilizzati nell'implementazione del linguaggio.

```
int foo(int*);[a]

int main() {
   int i, j, *p;[b]
   p = &j;[c]
   i = foo(p);[d]
   return 0;
}

int foo(int* q) {
   int r = 30;[e]
   *q = r;[f]
   return r;[g]
}
```

[a] dichiarazione della subroutine `foo`

[b] due variabili intere `i` e `j` e un puntatore ad intero `p`

[c] `*p` è un alias di `j`

[d] si invoca `foo` con un puntatore a intero come argomento, foo ritorna un intero (`30`) che viene assegnato a `i`

[e] `r` è una variabile locale al frame di `foo`, anche `q` è un simbolo locale al frame di `foo` ed è un alias di `p` in `main`

[f] si modifica il contenuto della locazione di memoria puntata da `q` che corrisponde alla variabile `j` nel frame di `main`

[g] terminata l'esecuzione `foo` ritorna `r` che vale `30`

3.15.2 Puntatori e vettori

Quando i valori che si devono passare a una subroutine sono vettori o matrici, allora, in C, non c'è nessuna possibilità di utilizzare un **vero** passaggio per valore.

In questo caso, anziché generare una copia del vettore o della matrice all'interno dello scope della subroutine, viene **sempre** generato un **alias** che consente di manipolare direttamente il vettore o la matrice definiti nel frame del chiamante.

In tutti i casi in cui si utilizza un vettore o una matrice come parametro, all'interno della subroutine si va a modificare il contenuto di ciò che è stato dichiarato in un altro frame. Anche in questo caso la motivazione va ricercata

nella necessità di contenere l'utilizzo della memoria: seguendo questa strategia non si replica mai la memoria allocata per il vettore o la matrice.

La dichiarazione di una subroutine che ha un vettore come argomento non è diversa da una dichiarazione di subroutine che ha come parametri semplici variabili. Supponiamo, ad esempio, di voler scrivere una subroutine che deve calcolare il massimo di un vettore monodimensionale di interi positivi, la sua dichiarazione sarà:

```
int vectmax(int[]);
```

L'argomento è un vettore di interi e il valore di ritorno, anch'esso intero, è il massimo valore contenuto nel vettore.

Adesso dobbiamo definire la subroutine e ci troviamo di fronte ad un bivio: il numero di elementi del vettore **è definito** e quindi, nel caso, dovremo definire una subroutine diversa per ogni dimensione, **oppure no**, e in quel caso, come faccio a saperne la dimensione?

In effetti il vettore, in questo caso (ma queste considerazioni valgono in generale), è un generico vettore di interi, dato che, dal punto di vista semantico il costrutto `a[]`, che sta ad indicare l'intero vettore, è equivalente a `*a`, ovvero è un puntatore alla prima posizione del vettore in memoria La notazione `a`, che viene utilizzato quando il vettore viene passato come parametro è invece equivalente a `&a[0]`, l'indirizzo della prima posizione del vettore in memoria. Questo perché, come abbiamo detto, non si può evitare di utilizzare un alias per il passaggio del vettore come parametro.

Quindi, a meno che non sia chiara dal contesto o non sia convenzionalmente definita, la soluzione è quella di aggiungere un parametro alla funzione: la **dimensione del vettore**.

```
int vectmax(int[],int);
```

Adesso la subroutine ha due argomenti: un vettore di interi e la sua dimensione, mentre il valore di ritorno ovviamente non varia. A questo punto la definizione della funzione potrà utilizzare il secondo parametro per la scansione del vettore:

```
int vectmax(int a[], int n) {
   int max = 0, i;
   for ( i=0 ; i<n ; i++ )
      if ( a[i] > max )
         max = a[i];
   return max;
}
```

È abbastanza chiaro come questa sia l'unica soluzione per definire una funzione generica per calcolare il massimo di un vettore di interi positivi di

lunghezza qualsiasi. Purtroppo **non esiste** una soluzione che consenta di evitare di passare **esplicitamente** la dimensione del vettore, dato che la dimensione con cui è stato dichiarato il vettore non viene memorizzata con esso nella tavola dei simboli.

Questione di forma

Un volta dichiarata e definita, la funzione `vectmax` può essere utilizzata con gli opportuni parametri:

```
int j;
int a[10];
...
j = vectmax(a,10);
```

La chiamata di funzione, in maniera totalmente equivalente, può essere anche scritta come:

```
j = vectmax(&a[0],10);
```

così come la definizione della funzione può avere come *signature*:

```
int vectmax(int *a, int n)
```

3.15.3 Puntatori e valori di ritorno

Se la subroutine che stiamo progettando deve calcolare più di un valore, ad esempio, anziché solo il massimo, sia il massimo che il minimo di un vettore, il meccanismo che abbiamo visto di utilizzare un solo valore di ritorno non funziona più.

Altri linguaggi, come Algol W, risolvono questo problema prevedendo una tipologia di passaggio di parametri per **risultato**. Questi parametri, al contrario di quelli passati per valore, che sono gli argomenti della funzione, sono destinati al passaggio dei parametri dal programma chiamato al programma chiamante. Potendone utilizzare tanti quanti ne vogliamo in una subroutine, superiamo il vincolo di ritornare uno ed un solo valore di ritorno.

In C si aggira il problema di non avere il passaggio per risultato fornendo alla subroutine tanti puntatori per quanti sono i valori che si vuole che ritorni. Questi, utilizzati come alias nel frame della subroutine, permettono di restituire più di un valore di ritorno.

Se la funzione che progettiamo deve calcolare sia il minimo che il massimo di un vettore avrà *signature*:

```
void vectminmax(int[],int,int*,int*);
```

mentre la sua definizione sarà:

```
void vectminmax(int a[], int n, int* min, int* max) {
   int i;
   *min = *max = a[0];
   for ( i=0 ; i<n ; i++ ) {
      if ( a[i] > *max ) *max = a[i];
      if ( a[i] < *min ) *min = a[i];
   }
   return;
}
```

Un esempio di invocazione di questa funzione è:

```
int larger, smaller, a[10];
...
vectminmax(a,10,&smaller,&larger);
```

3.16 Stringhe

I computer nascono per l'elaborazione dei numeri, ovvero per risolvere problemi che richiedono di trattare numeri per ottenere numeri. Ben presto, però, ci si è posti il problema di poter rappresentare anche delle informazioni **non numeriche** e questo ha portato a creare dei tipi che non sono strettamente legati al mondo dei numeri.

Abbiamo visto che un tipo primitivo di una gran maggioranza di linguaggi per la rappresentazione di informazioni non numeriche è il tipo **carattere** (`char` in C). Avere la possibilità di utilizzare solo un carattere alla volta non è né comodo né utile ed è quindi opportuno poter trattare delle intere **parole** o **frasi** come oggetti in un programma: queste si chiamano **stringhe** (dall'inglese *string*).

3.16.1 Linguaggi per l'elaborazione del testo

Nel corso dell'evoluzione dei linguaggi di programmazione si sono sviluppati anche linguaggi **specializzati** nel trattamento del testo, ovvero che possiedono delle funzionalità molto potenti per compiere operazioni su parole e frasi.

Sono perlopiù **linguaggi di scripting**, quali Perl, Python o Ruby, che consentono di operare in maniera molto flessibile ed efficace sulle stringhe utilizzando le **espressioni regolari**, un formalismo proposto da Kleene negli anni '50 ed oggi divenuto lo standard per la descrizione di porzioni semplici dei linguaggi formali.

Nei linguaggi le stringhe possono essere:

1. Array di caratteri (è la soluzione più semplice, non richiede sforzi aggiuntivi di progettazione e implementazione);

2. Avere una considerazione ed un supporto linguistico speciale.

In C sono **array di caratteri**, mentre nei suoi discendenti orientati agli oggetti sono un tipo predefinito del linguaggio (una classe).

3.16.2 Stringhe in C

La scelta di implementare le stringhe come array di caratteri è una conseguenza diretta delle due osservazioni seguenti:

1. Un carattere è un tipo predefinito del linguaggio;
2. Una stringa è una successione ordinata di caratteri.

Utilizzando array di caratteri per rappresentare stringhe non esisteva la necessità di definire un nuovo tipo e si potevano sfruttare le caratteristiche degli array.

Questa semplicità di progettazione ed implementazione si paga al momento dell'utilizzazione:

1. Le stringhe sono array, hanno dimensione statica, sono allocate nel frame della subroutine (non nello heap) e quindi non è possibile allungare ed accorciare una stringa a run-time con supporto da parte dell'ambiente a run-time;
2. Il controllo di fine stringa (che eredita il controllo di dimensione dei vettori) è lasciato al programmatore.

Letterali di tipo stringa

C e i suoi discendenti distinguono tra letterali[14] di tipo carattere che sono racchiusi tra singoli apici (`'a'`) e letterali di tipo stringa, che sono racchiusi tra doppi apici (`"ma va'?"`).

In altri linguaggi, quali il Pascal, invece, un carattere non è altro che una stringa di lunghezza 1, perché nel linguaggio si sono progettate prima le stringhe dei caratteri.

I letterali di tipi stringa sono molto particolari, perché sono delle espressioni costanti che non corrispondono ad alcun tipo del linguaggio, sono una eccezione rispetto a tutto quanto abbiamo visto sinora. Che le stringhe abbiano uno status speciale, con un trattamento differenziato rispetto a tutte le altre collezioni di elementi uguali raccolte in array, è spesso vero nella maggior parte dei linguaggi per la natura stessa delle stringhe.

[14] Il termine letterale definisce degli oggetti immutabili durarnte l'esecuzione del programma, è lo stesso che, in matematica, si definisce costante.

Dichiarazione di stringhe

Dichiarare una variabile di tipo stringa, quindi, in C, non è possibile, si devono utilizzare array di caratteri la cui dichiarazione, utilizzando la sintassi standard, sarebbe

```
char str[11] = {'M','a','n','n','a','g','g','i','a','!','\0'}
```

Da notare l'esplicito utilizzo del carattere di fine stringa, il carattere di *escape* `\0`, che deve essere inserito a cura del programmatore, pena il non riconoscimento della terminazione della stringa.

Fortunatamente si può utilizzare una maniera più compatta, in cui si sfrutta anche la possibilità del compilatore di calcolare automaticamente la dimensione dell'array in base alla lista dei valori di inizializzazione.

```
char str[] = "Mannaggia!"
```

Per complicare le cose, in questo caso, il carattere di fine stringa non è richiesto, viene inserito dal compilatore e l'array avrà una dimensione aumentata di 1 rispetto alla lunghezza della stringa.

Costanti stringa

Un'altra peculiarità delle stringhe è la modalità con cui vengono gestite le costanti. Una costante di tipo carattere, ad esempio, può essere dichiarata utilizzando la parola chiave `const` accoppiata ad una dichiarazione di variabile:

```
const char urka='!';
```

Dato però che non possono esistere variabili di tipo stringa, l'unico modo per avere a disposizione una costante stringa è dichiarare un puntatore ad un letterale:

```
char* k = "Che bello il C!";
```

La conseguenza, non banale, è che il puntatore può essere riassegnato durante l'esecuzione, (anche nel caso in cui si faccia precedere dal modificatore `const`) perdendo l'indirizzo della porzione statica di memoria che contiene il letterale!

Operazioni sulle stringhe

Utilizzando le stringhe si è portati a dimenticare una regola sintattica basilare che è, invece, opportuno ricordare: un array non può essere un *lvalue*.

Operando con gli array non è possibile, se non in fase di inizializzazione, assegnare con una singola operazione l'intero array, quindi, anche nel caso di stringhe, non è possibile effettuare (supponendo che `str` sia un array di `char`) un'operazione quale:

```
str = "No, no, non si puo' fare...";
```

e neanche

```
str1 = str2;
```

Con `str1` e `str2` array di `char`.

3.16.3 La libreria `string`

La filosofia di base del linguaggio C consiste nel non avere istruzioni particolari che consentano di risolvere problemi specifici (*kernel* compatto del linguaggio) e demandarne la soluzione a funzioni di libreria.

Nel caso delle stringhe le funzioni disponibili si trovano nella libreria di manipolazione delle stringhe[15] e consentono di effettuare, tra le altre:

1. Copia di stringhe (`strcpy`);
2. Concatenazione di stringhe (`strcat`);
3. Confronto tra stringhe (`strcmp`);
4. Operazioni di ricerca di caratteri o sottostringhe all'interno di una stringa (`strchr`, `strstr`).

Le funzioni di libreria consentono di effettuare le operazioni di base sulle stringhe, se si devono effettuare operazioni più complesse e articolate forse è il caso di cambiare linguaggio di programmazione!

3.17 Collezioni di dati eterogenei

I tipi di dati noti come **record** permettono di immagazzinare e manipolare insieme tipi eterogenei di dati correlati tra loro. Alcuni linguaggi (in particolare Algol 68, C, C++, e Common Lisp) utilizzano il termine **struttura** (dichiarata con la parola chiave `struct`), invece di record. Fortran 90 chiama semplicemente i suoi record **types**: sono l'unica forma di tipo definito dal programmatore oltre agli array, che hanno una loro propria sintassi. Un linguaggio object-oriented come Java non possiede la nozione di struttura: per tutti i tipi di dati si utilizzano classi ed oggetti che possono essere visti come un altro modo, più complesso e potente, di raggruppare dati eterogenei.

Per capire la potenza dell'utilizzo di questi tipi di dati, partiamo da un semplice esempio.

Proviamo a ipotizzare di dover organizzare le informazioni che possono servire per modellare un elemento chimico all'interno del nostro ambiente di

[15] Le dichiarazioni si trovano nell'header `string.h`.

programmazione, per decidere se possano essere una collezione di dati omogenei o eterogenei. Sicuramente vogliamo che l'elemento che modelliamo abbia un **nome**, quello che viene comunemente utilizzato per l'elemento (idrogeno, ossigeno, ecc.): il nome è una stringa di caratteri.

Poi vogliamo che, assieme al nome, vengano memorizzate le principali caratteristiche dell'elemento, come il suo **numero atomico** e il **peso atomico** che possono essere, rispettivamente, un intero e un reale.

Magari vogliamo che venga anche segnalata una proprietà dell'elemento quale quella di essere o meno un **metallo**, che può essere rappresentata con un valore booleano.

Abbiamo descritto una collezione di dati tutti diversi tra loro, accomunati dalla proprietà di descrivere lo stesso oggetto. Un vettore non serve sicuramente al nostro scopo, dobbiamo ricorrere a un record.

In C, un record che descriva un elemento chimico con queste proprietà si chiama `struct` e possiamo definirlo come:

```
struct element {
   char name[2];
   int atomic_number;
   double atomic_weight;
   _Bool metallic;
}
```

La stessa dichiarazione, tradotta in Pascal, sarebbe:

```
type two_chars = packed array [1..2] of char;
type element = record
   name : two_chars;
   atomic_number : integer;
   atomic_weight : real;
   metallic : Boolean
end;
```

Come si può vedere, sono molto simili. La differenza principale consiste nella possibilità di Pascal di dichiarare nuovi tipi come vettori di tipi esistenti (in questo caso un array di due caratteri) e l'esistenza di un vero tipo booleano. Questo rende più controllata la costruzione del tipo composto.

3.17.1 Le strutture in C

I campi di un record

Ogni componente di un record si chiama **campo** (***field***).

Per far riferimento ad un dato campo di un record molti linguaggi, C compreso, utilizzano l'operatore **punto** `.`.

```
struct element copper;
const double AN = 6.022e23; /* num. di Avogadro */
...
copper.name[0] = 'C'; copper.name[1] = 'u';
double atoms = mass / copper.atomic_weight * AN;
```

Il nome del record non è automaticamente un nuovo tipo, infatti per utilizzarlo dobbiamo farlo precedere dalla parola riservata `struct`.

La sintassi del linguaggio non ci **impone** di creare un nuovo tipo ogni volta che utilizziamo una struttura, dobbiamo specificarlo esplicitamente.

Strutture come tipi

Comunemente, quindi, si accoppia la definizione di un nuovo tipo alla definizione di una nuova struttura.

```
typedef struct element {
...
} element;
```

In questo modo è possibile utilizzare direttamente il tipo `element` per dichiarare variabili che abbiano l'organizzazione della struttura:

```
element iron;
```

I due simboli (nome della struttura e nome del nuovo tipo) è lecito che siano uguali poiché i due spazi di nomi sono separati, ma, quando si associa ad un nuovo tipo, il nome della struttura è pressoché sempre omesso e la prassi prevede che il nuovo tipo abbia iniziale maiuscola:

```
typedef struct {
...
} Element;
```

Assegnamento tra strutture

I record, insieme agli array, sono i tipi composti standard di un linguaggio, la differenza sta nell'eterogeneità dei record contrapposta all'omogeneità degli array. In C però una differenza **sostanziale** tra i due tipi composti è la possibilità per le strutture di essere *lvalue*.

È infatti ammesso dal linguaggio effettuare un'operazione come:

```
Element copper, another_copper;
...
another_copper = copper;
```

Il compilatore traduce l'operazione di assegnamento tra strutture in un'operazione di assegnamento campo per campo.

Strutture come parametri

Le strutture possono essere utilizzate all'interno del programma anche come parametri di una subroutine o come tipo di ritorno di una funzione:

```
Element philosopher_stone(Element);
```

Dopo aver definito la funzione possiamo invocarla con:

```
gold = philosopher_stone(lead);
```

Dato che il passaggio dei parametri avviene per valore, sia in invocazione che in ritorno, la struttura viene copiata, campo per campo dal chiamante al chiamato e viceversa.

Strutture e vettori

Se proviamo a sostituirci al compilatore nella traduzione dell'assegnamento tra strutture in un'operazione di assegnamento campo per campo dovremmo riscrivere esplicitamente l'assegnamento come:

```
another_copper.name = copper.name;
another_copper.atomic_number = copper.atomic_number;
another_copper.atomic_weight = copper.atomic_weight;
another_copper.metallic = copper.metallic;
another_copper = copper;
```

C'è qualcosa di sintatticamente scorretto in questo assegnamento perché abbiamo utilizzato il campo `name`, che è un array, come *lvalue*. Nella nostra operazione di traduzione dovremmo quindi scrivere:

```
another_copper.name[0] = copper.name[0];
another_copper.name[1] = copper.name[1];
```

La scelta del linguaggio è ancora una volta guidata dalla impossibilità di controllare la dimensione degli array. Nel caso degli array, non potendo avere un controllo linguistico sulla dimensione dei due vettori sul lato destro e sinistro di un'operazione di assegnamento, non si consente di poterla effettuare. Nel caso di una struttura, dato che c'è la **sicurezza** che le due strutture, sul lato destro e sinistro, siano uguali in dimensione allora l'assegnamento è consentito.

L'effetto secondario è che anche campi di strutture che sono vettori possono essere utilizzati dato che, comunque, in entrambe le strutture hanno la stessa dimensione!

Annidamento di strutture

Nella progettazione di strutture si possono utilizzare, come campi, tipi composti, e quindi, oltre che array, anche altre strutture. Possiamo, ad esempio, progettare una struttura destinata a contenere il nome di una persona:

```
#define NAME_LEN 20
typedef struct{
   char first[NAME_LEN+1];
   char last[NAME_LEN+1];
} Name;
```

In un secondo tempo, questa struttura, può essere utilizzata in qualsiasi altra struttura in cui devo avere un campo che memorizza il nome di una persona.

Costruisco la mia progettazione in maniera **modulare**, cercando di avere il maggior riutilizzo possibile degli elementi già progettati.

Per progettare la struttura che contiene le informazioni relative ad uno studente posso quindi riutilizzare la struttura che definisce un nome:

```
typedef struct{
   Name name;
   int id;
   int age;
   char sex;
} Student;
```

Inizializzazione di strutture

Quando dichiaro una variabile di tipo struttura (come `Student` nell'esempio) posso inizializzarla con una lista di inizializzazione che segue la stessa sintassi, con contenuto opportuno, delle liste degli array:

```
Student matteo = {{"Matteo", "Renzi"},23432,39,'m'};
```

Notare l'annidamento della lista di inizializzazione di `name` all'interno di quella di `Student`.

Accesso ai campi di strutture annidate

L'operatore `.` di accesso ai campi di una struttura deve essere utilizzato in maniera iterativa quando si voglia accedere ai campi di strutture annidate. Nell'esempio precedente, se voglio accedere al nome di battesimo dello studente i cui dati sono accessibili tramite la variabile `matteo` devo utilizzare il costrutto:

```
printf("%s",matteo.name.first);
```

L'operatore `.` ha il livello di priorità più alta tra tutti gli operatori in C ed è associativo a sinistra per cui quando si valuta l'espressione secondo parametro della subroutine `printf`, come primo passo si valuterà `matteo.name` e si utilizzerà il risultato per valutare `.first`.

Array di strutture

Una struttura, una volta definita come nuovo tipo, può essere utilizzata oltre che come campo di un'altra struttura, anche come elemento con il quale si possono costruire array:

```
Student class[2] = {{"Matteo", "Renzi"},23432,39,'m'},
                   {{"Giorgio", "Napolitano"},19891,89,'m'}};
```

Adesso l'accesso ad un elemento dell'array può essere seguito da un accesso ad un campo della struttura e così via in cascata:

```
printf("%s",class[0].name.first);
```

Gli operatori `.` e `[]` hanno lo stesso livello di priorità e sono entrambi associativi a sinistra, quindi, la valutazione dell'espressione come primo passo valuterà `class[0]`, poi `.name` e infine `.first`.

Puntatori e strutture

Una struttura occupa una porzione di memoria sufficiente per contenere tutti i dati dei campi che la compongono, quindi possiamo definire dei puntatori, di tipo opportuno, che possono essere utilizzati per accedere direttamente a quella porzione della memoria:

```
Student *best;
```

Questi puntatori possono essere utilizzati esattamente nello stesso modo in cui si utilizzano puntatori a tipi semplici. Ad esempio, per simulare il passaggio di parametro per riferimento ad una funzione. Ovviamente continua a valere la regola per cui una variabile (anche di tipo puntatore) è inutilizzabile sinché non viene assegnata:

```
best = &class[1];
```

L'operatore –> Per accedere alla struttura riferita dal puntatore si può utilizzare la sintassi standard dei puntatori, con l'uso dell'operatore di indirezione `*`:

```
printf("%s",(*best).name.last);
```

Dato che l'uso di puntatori per l'accesso a strutture è un'operazione molto diffusa, i progettisti del linguaggio hanno previsto di definire un operatore specifico (`->`) che, pur essendo solo zucchero sintattico rende questo accesso più semplice da gestire:

```
printf("%s",best->name.last);
```

Un enorme vantaggio nell'uso dell'operatore `->` sta nell'evitare errori di precedenza, dato che l'operatore di indirezione ha **priorità minore** rispetto all'operatore di accesso ai campi.

3.18 Allocazione dinamica

È sempre necessario inquadrare storicamente lo sviluppo dei linguaggi di programmazione per comprendere le motivazioni che stanno alla base delle **scelte di progettazione**.

Tra la fine degli anni '60 e l'inizio degli anni '70, quando si è progettato e iniziato ad utilizzare il linguaggio C, il rapporto tra memoria disponibile sulla macchina e memoria che il programma si poteva permettere di utilizzare era molto alto. PDP-11, la macchina su cui è stato inizialmente sviluppato C, aveva, nella sua versione iniziale (primavera 1970), **56 KBytes** di memoria centrale. È abbastanza facile comprendere come la memoria fosse un bene estremamente prezioso, la cui gestione poteva anche risultare molto complessa ed articolata, purché si risparmiasse al massimo sul suo utilizzo.

La necessità di garantire la massima modularità nell'allocazione (e consecutiva utilizzazione) della memoria disponibile sullo heap ha portato a sviluppare svariati metodi che consentono di gestire questo compito.

I principali requisiti nella gestione dello heap sono:

1. Occupare lo **spazio** disponibile nella migliore maniera possibile;
2. Impiegare il minor **tempo** possibile per svolgere i compiti necessari alla gestione dei blocchi di memoria.

È completamente oltre i nostri obiettivi entrare nel merito delle scelte di gestione della memoria che sono a carico del sistema operativo. Per i nostri scopi ci interessa solo sapere che in un linguaggio come C esistono degli strumenti linguistici (meglio, di libreria) che consentono di richiedere esplicitamente porzioni di memoria per utilizzarle all'interno del proprio programma.

3.18.1 Allocazione dinamica di memoria in C

I meccanismi di allocazione di memoria che abbiamo utilizzato sino ad ora sono **automatici**, l'utente non deve esplicitamente indicare, al compilatore o all'ambiente a run-time, quando e di quanta memoria ha necessità.

La memoria allocata **staticamente** viene riservata per gli oggetti in essa contenuti all'inizio dell'esecuzione utilizzando la taglia degli oggetti per definire la dimensione. Nel caso della memoria allocata sullo **stack** è il compilatore che provvede a calcolare la dimensione di ogni frame e, a run-time, a ogni invocazione di subroutine viene allocata una porzione di memoria di quella dimensione.

i meccanismi che consentono di utilizzare lo heap, in C, sono, invece, **espliciti**: l'utente definisce la taglia della memoria necessaria e la riserva con comandi appositi.

Allocazione di memoria: comandi

In C la gestione dinamica della memoria viene effettuata per mezzo di subroutine che sono contenute nella libreria standard[16].

Le subroutine che allocano memoria sono tre:

`void* malloc (size_t size)` alloca un blocco di `size` byte di memoria, ritornando un puntatore all'inizio del blocco; il contenuto non è inizializzato ed è quindi indeterminato;

`void* calloc (size_t num, size_t size)` alloca un blocco di memoria per un array di `num` elementi, ognuno dei quali di dimensione `size`, e inizializza tutti i bit del blocco a zero;

`void* realloc (void* ptr, size_t size)` la dimensione del blocco di memoria puntato da `ptr` è modificata per divenire `size` byte, aumentando o diminuendo la quantità di memoria disponibile nel blocco

I nomi delle subroutine sono dei richiami mnemonici per *memory allocation* (`malloc`), *clear on allocation* (`calloc`) e *reallocation* (`realloc`).

La funzione `calloc` può essere utilizzata anche per allocare memoria per oggetti diversi da array, ha come effetto di allocare **num** $\times$ **size** byte di memoria inizializzati a zero. Il tipo `size_t` è un accorgimento linguistico per definire un tipo generico di dimensione 1 byte.

La posizione all'interno dello heap in cui viene riservato il blocco di memoria che si richiede non è nota al programmatore, è una scelta dell'ambiente a run-time nel momento in cui si effettua la richiesta, per questo alla memoria allocata dinamicamente si può accedere solamente tramite puntatori.

Il valore di ritorno è, per tutte le funzioni, il puntatore al blocco di tipo generico `void*` di cui si deve effettuare un casting quando si utilizza.

La memoria non più utilizzata si **deve** deallocare esplicitamente con la procedura `void free (void* ptr)`, altrimenti rimane nella disponibilità del programma sino al termine dell'esecuzione[17].

[16] L'header che ne contiene le dichiarazioni è `stdlib.h` ed è una libreria che, per default, viene linkata a qualsiasi programma senza necessità di indicazioni esplicite da parte del programmatore.

[17] Ricordiamo infatti che C non dispone di alcun meccanismo di *garbage collection*.

Supponiamo di voler allocare la memoria necessaria per ospitare una stringa di una certa dimensione `n` e poi di volerla riempire di caratteri in maniera casuale:

```
int i,n = 25;
char* buffer;[a]
buffer = (char*) malloc (n+1);[b]
if (buffer==NULL) return (1);[c]
for (i=0; i<n; i++)[d]
   buffer[i]=rand()%26+'a';
buffer[n]='\0';[e]
printf("%s",buffer);[f]
free (buffer);[g]
```

[a] puntatore ad un'area di memoria che contiene `char`
[b] alloco n+1 byte e faccio casting a `char*`
[c] controllo se la memoria è effettivamente disponibile
[d] in ogni elemento dell'array si inserisce casualmente una lettera minuscola
[e] nell'ultimo elemento si inserisce il carattere di escape
[f] vediamo cosa ha prodotto!
[g] fatto! si cancella tutto …

La variabile `buffer` viene riservata nel frame di questa porzione di codice e, prima di essere utilizzata, deve puntare ad un blocco di memoria che possa contenere i caratteri. Da notare il suo doppio uso come puntatore e nome di array che segue la politica di unificazione di queste due entità sintattica di C.

Il simbolo `NULL`

Nell'esempio precedente abbiamo usato il confronto

```
if (buffer==NULL) return (1);
```

Il simbolo `NULL` è una macro, definita nell'header della libreria, e ha il significato di **puntatore nullo**, ovvero è un puntatore che non punta a nessun oggetto utilizzabile. Il valore di NULL è una scelta di implementazione, ma tipicamente si definisce come:

```
#define NULL ((void *)0)
```

Notare che un puntatore, nella sostanza, non è **mai** nullo, solo convenzionalmente si assume che in questo caso la locazione di memoria puntata (con indirizzo `0`) non sia realmente utilizzabile.

L'operatore `sizeof`

Il programmatore che deve allocare memoria per tipi di dati diversi tra loro non ha la possibilità di conoscere quanta memoria deve essere disponibile per

ogni tipo di dato che utilizza. A questo scopo il linguaggio mette a disposizione l'operatore unario `sizeof`:

```
int* pointer = malloc(sizeof(int) * NUMPOS);
```

L'operatore `sizeof` ha come operando un tipo di dati, di cui restituisce, come risultato dell'operazione, la dimensione in bytes. Il risultato dell'istruzione dell'esempio precedente è quello di allocare lo spazio necessario per un array di interi di `NUMPOS` posizioni che potrà poi essere utilizzato con il meccanismo di indicizzazione proprio degli array:

```
for (i=0 ; i<NUMPOS ; i++)
   pointer[i] = i*i;
```

3.18.2 Gestione delle stringhe

Come abbiamo visto, in C, le stringhe sono array di caratteri e, di conseguenza, gli array di stringhe sono matrici di caratteri. Un array di stringhe, che contiene tutti i nomi dei giorni della settimana in inglese, si può dichiarare come:

```
char days[][10] = {
   "Sunday", "Monday", "Tuesday", "Wednesday",
   "Thursday", "Friday", "Saturday"
};
```

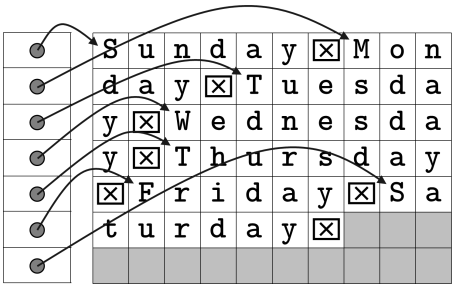
L'allocazione di memoria che ne consegue è:

S	u	n	d	a	y	☒			
M	o	n	d	a	y	☒			
T	u	e	s	d	a	y	☒		
W	e	d	n	e	s	d	a	y	☒
T	h	u	r	s	d	a	y	☒	
F	r	i	d	a	y	☒			
S	a	t	u	r	d	a	y	☒	

Una maniera alternativa di generare gli stessi oggetti, che, a prima vista, potrebbe sembrare esattamente uguale alla precedente, passa per l'uso esplicito di puntatori:

```
char* days[] = {
   "Sunday", "Monday", "Tuesday","Wednesday",
   "Thursday", "Friday", "Saturday"
};
```

È ancora possibile accedere ogni singolo carattere di ogni stringa con doppia indicizzazione (es. `days[2][3]`) ma adesso l'allocazione di memoria è:



3.18.3 Puntatori e allocazione dinamica

Utilizzando la seconda modalità di allocazione possiamo anche dividere la definizione dell'array di stringhe in due fasi:

1. Dichiarare l'array di puntatori a caratteri (stringhe);
2. Allocare dinamicamente la memoria necessaria per ogni stringa e collegarla al puntatore.

Dichiaro l'array di stringhe:

```
char* days[7];
```

Ad ogni puntatore assegno una stringa:

```
days[0] = (char*) malloc(7);
strcpy(days[0],"Sunday");
days[1] = (char*) malloc(7);
strcpy(days[1],"Monday");
...
```

Riallocazione

Il vantaggio dell'allocazione dinamica, l'unico vantaggio, è quello di consentire di variare la dimensione degli oggetti che si sono utilizzati nel momento in cui se ne ha la necessità.

Se, ad esempio, vogliamo variare i nomi dei giorni della settimana, sostituendo i nomi in inglese con i nomi corrispondenti in italiano possiamo utilizzare la riallocazione di memoria:

```
days[0] = (char *) realloc(days[0],9);
strcpy(days[0],"Domenica");
days[1] = (char *) realloc(days[1],8);
strcpy(days[1],"Lunedi'");
...
```

È importante che il puntatore ritornato dalla funzione **realloc** venga utilizzato per riassegnare la locazione di memoria che, in seguito all'espansione o riduzione del blocco allocato, può essere stata modificata.

3.18.4 Puntatori a puntatori

Il concetto di puntatore è semplice: è una variabile che non contiene un valore ma un **indirizzo della memoria** nel quale si può trovare un valore.

Dopo aver così definito i puntatori, in effetti, diventano essi stessi dei particolari valori che possono essere memorizzati in locazioni di memoria appropriate. È quindi possibile definire dei puntatori a valori che sono essi stessi dei puntatori, aumentando il livello di **indirezione** da un passo a due passi.

Questo procedimento, in teoria, può essere iterato per un numero qualunque di passi, ma non è affatto consigliato farlo, dato che la complessità della gestione del codice, e la probabilità di errore, aumenta ad ogni passo di indirezione.

Supponiamo, ad esempio, di voler definire un poligono, il cui numero di vertici non è noto a priori. Per prima cosa definiamo un nuovo tipo di dato che consenta di memorizzare in una struttura le due coordinate di un punto:

```
typedef struct {
   float x;
   float y;
} Point;
```

Non sapendo, al momento della dichiarazione, da quanti elementi sarà costituito l'array di `Point` che contiene i vertici del poligono lo dichiariamo come:

```
Point** coordinates;
```

Il simbolo `coordinates` è un puntatore a puntatore a `Point`.

Nel momento in cui vogliamo utilizzare `coordinates` come array di puntatori a `Point` per prima cosa dobbiamo allocare la memoria necessaria per l'array stesso (di dimensione `n`), in cui ogni elemento è un puntatore:

```
coordinates = (Point**) malloc(sizeof(Point*)*n);
```

Per calcolare la dimensione di ciascun elemento del vettore viene utilizzato l'operatore `sizeof` che consente di ottenere la dimensione di un tipo o di un'espressione (in questo caso di un tipo). Vale la pena di ricordare che la dimensione dei puntatori è sempre la medesima, qualunque sia il tipo a cui puntano, ma per una questione di forma è sempre opportuno utilizzare il tipo adeguato.

Adesso `coordinates` fa riferimento ad un blocco di memoria che contiene `n` puntatori consecutivi ad oggetti di tipo `Point`. Dovrebbe essere chiaro a questo punto che l'array non è utilizzabile fintanto che non si allochino gli elementi a cui i puntatori facciano riferimento.

```
coordinates[i] = (Point*) malloc(sizeof(Point));
coordinates[i]->x = ...
coordinates[i]->y = ...
```

L'accesso ad ogni elemento avviene tramite due meccanismi:

1. Ogni elemento del blocco viene indicizzato utilizzando il meccanismo di accesso agli array;
2. Per accedere ai campi della struttura si utilizza l'operatore `->` dato che ogni elemento di `coordinates` è un puntatore alla struttura.

3.19 Tipi ricorsivi

Un **tipo ricorsivo** è un tipo i cui oggetti possono contenere uno o più **riferimenti** a oggetti dello stesso tipo. È assai comune che i tipi ricorsivi siano record (strutture nella nomenclatura C) dato che è normale che, oltre al riferimento contengano altre informazioni aggiuntive di altro tipo rispetto al puntatore.

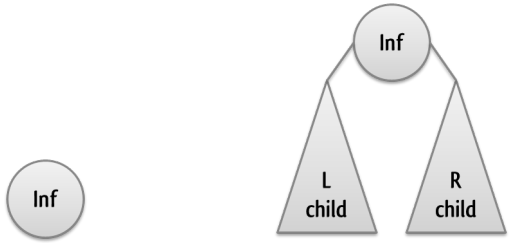
I tipi ricorsivi sono usati per costruire una grande varietà di strutture dati *linkate* quali, ad esempio, le liste e gli alberi. Questi tipi sono molto semplici da creare in linguaggi che usano un modello di variabili basato sul **riferimento** (es. Lisp, ML o Java). In linguaggi come C o Pascal che usano un modello di variabili basato sul **valore** i tipi ricorsivi si devono costruire utilizzando esplicitamente il concetto di puntatore.

3.19.1 Un esempio: gli alberi binari

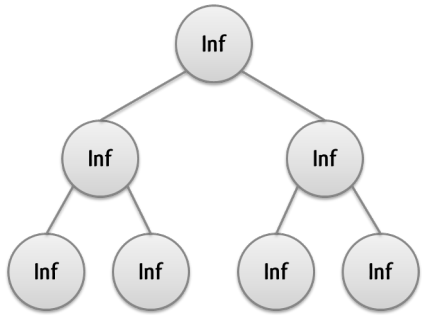
Gli alberi sono le strutture dati più pervasive nel campo dell'informatica. Si adoperano alberi di vari tipi in quasi tutte le applicazioni che richiedono di classificare oggetti in maniera ordinata e di recuperarli efficientemente.

Un **albero binario** è una struttura dati in cui ogni nodo ha uno o, al più, due figli. La struttura può essere definita ricorsivamente in maniera molto semplice:

Un nodo è un albero binario
Un nodo con uno o due sottoalberi binari come figli è un albero binario



Descritto in maniera completa, un albero binario si presenta come un insieme di nodi legati da relazioni di **parentela**, padre-figlio:



Il tipo ricorsivo albero binario

Per costruire il tipo di dato che possa ben rappresentare un albero binario dobbiamo sostanzialmente pensare a come rappresentare un suo nodo. Un elemento deve contenere:

- Le informazioni relative al dato contenuto nel nodo;
- I riferimenti che consentono di **collegare** il nodo a entrambi i figli.

La soluzione più ovvia è quella di costruire l'elemento come un record che contiene l'informazione (può essere un intero o qualsiasi altro tipo di dato) e due riferimenti a due elementi dello stesso tipo del record che stiamo definendo:

In Pascal questo tipo di dato, se volessimo utilizzare caratteri come informazione contenuta in ogni nodo, si dichiara nel seguente modo:

```
type bintreeptr = ^ bintree;
   bintree = record
      val : char;
      left, right : bintreeptr
   end;
```

È da notare che in Pascal si deve esplicitamente dichiarare un nuovo tipo per ogni puntatore.

La stessa dichiarazione in C, ma questa volta per costruire una albero che organizza interi anziché caratteri, è molto simile:

```
struct bintree {
   int val;
   struct bintree *left, *right;
};
```

Come sempre, se vogliamo semplificare l'utilizzo della nuova struttura è più conveniente definire un nuovo tipo che la incapsula:

```
typedef struct bintree {
   int val;
   struct bintree *left, *right;
} BinTree;
```

Il riferimento interno alla dichiarazione di struttura **non può essere** al tipo `BinTree` perché, non essendone ancora stata effettuata la dichiarazione, non è ancora un tipo disponibile nel sistema di tipi del linguaggio. È quindi **indispensabile** utilizzare un nome anche per la struttura in se, dato che il riferimento interno è **esattamente** a quel nome.

Dopo la dichiarazione precedente, per utilizzare un oggetto di tipo albero binario nel nostro programma si possono dichiarare delle variabili di tipo `BinTree`:

```
BinTree tree;
```

o dei puntatori ad oggetti di tipo `BinTree`:

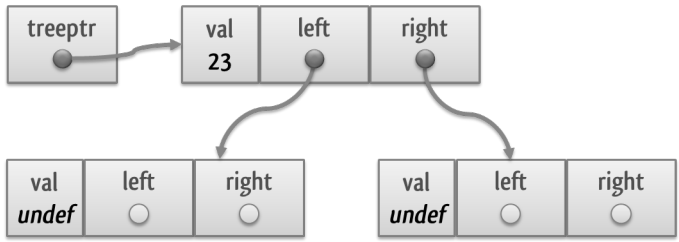
```
BinTree *treeptr;
```

In questo secondo caso si utilizzerà poi il puntatore in abbinamento all'allocazione dinamica di memoria:

```
treeptr = (BinTree *) malloc(sizeof(BinTree));
```

A seconda della modalità di allocazione che utilizzo, i vari campi della struttura vengono poi acceduti utilizzando gli operatori `.` e `->`. Ad esempio, se utilizzo puntatori e allocazione dinamica:

```
treeptr->val = 23;
treeptr->left = (BinTree*) malloc(sizeof(BinTree));
treeptr->right = (BinTree*) malloc(sizeof(BinTree));
```



Capitolo 4

Complessità computazionale

4.1 Introduzione

Per ogni problema dato, di solito, è possibile determinare non uno, ma un insieme di algoritmi in grado di risolverlo. Quali criteri si possono adottare per scegliere l'algoritmo da utilizzare?

Sicuramente l'algoritmo deve garantire la correttezza formale della risoluzione del problema, ma si può anche prendere in considerazione il tempo di esecuzione medio che quell'algoritmo impiega per completare le operazioni e le risorse necessarie all'esecuzione. Solitamente, il metro di valutazione universalmente adottato per decidere quale algoritmo è più efficiente di un altro, è quello della **complessità computazionale**, che può essere riassunto come il tempo necessario all'esecuzione dell'algoritmo su un generico computer con architettura tradizionale dotata di una sola CPU.

Supponiamo di dover risolvere il problema del calcolo della somma dei primi n numeri naturali, è possibile utilizzare due diversi algoritmi, qui tradotti in codice:

Calcolo iterativo

```
int i, n, sum=0;
...
for (i=1; i <= n; i++)
    sum += i;
```

Formula di Gauss

```
int n, sum=0;
...
sum = (n * (n+1))/2;
```

Quale delle due soluzioni conviene scegliere? Valutiamole dai punti di vista che, abbiamo capito, sono necessari per fare una scelta sensata.

- **Correttezza**: ambedue gli algoritmi sono corretti, ossia risolvono il problema e arrivano alla medesima soluzione;
- **Risorse**: il primo algoritmo richiede 3 variabili intere, il secondo invece soltanto 2, la differenza non è particolarmente significativa;
- **Numero di istruzioni**: il primo algoritmo effettua un aggiornamento di una variabile di somma per n volte (vale a dire n letture dalla memoria, n somme ed n scritture in memoria), mentre il secondo algoritmo effettua un unico calcolo, composto da una moltiplicazione, una somma, una divisione ed un assegnamento.

Appare evidente come il secondo algoritmo effettui molte meno operazioni del primo. Questo è indubbiamente un vantaggio di cui tenere conto, soprattutto nel caso in cui n sia un numero piuttosto grande. Vedremo nel seguito come si possa formalizzare il calcolo della complessità computazionale per definire realmente quale dei due metodi sia il più efficiente.

La valutazione della complessità computazionale di un algoritmo deve soddisfare due requisiti:

1. **Non deve dipendere da una particolare macchina, né da un particolare compilatore (o da particolari ottimizzazioni del codice)**. Questo perché il tempo di esecuzione varia in relazione a fattori indipendenti dall'algoritmo, come ad esempio la velocità di calcolo del processore, e perché le istruzioni effettivamente eseguite dal processore dipendono dalla traduzione effettuata dal compilatore;
2. **Deve essere espressa come funzione dei dati in ingresso**. Questo perché un algoritmo elabora un insieme dati in ingresso per arrivare a una soluzione (dati in uscita); quello che solitamente interessa è determinare in che modo i dati in ingresso (la *quantità* di dati o i *valori* da essi assunti) va ad influenzare il tempo di esecuzione dell'algoritmo stesso.

Per ciascun algoritmo, inoltre, vengono solitamente studiati i comportamenti in diversi casi:

1. Caso **migliore**: il caso in cui l'algoritmo giunge a una soluzione nel tempo minimo possibile;
2. Caso **pessimo**: il caso in cui l'algoritmo giunge a una soluzione nel tempo massimo possibile;
3. Caso **medio**: il caso in cui l'algoritmo giunge a una soluzione nel tempo *medio*.

La relazione tra il tempo di esecuzione di un algoritmo t e la dimensione dei dati in ingresso n (dove n può rappresentare la mole dei dati o il valore assunto da essi a seconda del problema) è espressa dalla funzione $T(n)$ e il calcolo della complessità computazionale consiste nel *determinare l'espressione* della funzione $T(n)$.

4.1.1 Perché valutare la complessità computazionale?

Lo studio della funzione $T(n)$ ha dei vantaggi pratici non indifferenti nella scelta di quale algoritmo implementare:

1. Fissata una dimensione di dati in ingresso, è possibile conoscere una stima attendibile dei tempi di esecuzione relativi ai diversi algoritmi presi in considerazione;
2. È possibile studiare come varia il tempo di ingresso in relazione al variare della dimensione dei dati, e valutare quale algoritmo ha una migliore *scalabilità*;
3. Fissato un limite massimo di tempo di esecuzione, è possibile determinare quale è la dimensione massima dell'input che può essere correttamente gestita entro i limiti temporali richiesti, e individuare quindi un *upper bound* dell'input.

4.2 Notazione

Per rappresentare la funzione $T(n)$ si utilizzano delle particolari notazioni che mettono in relazione l'input con il tempo di esecuzione:

1. $\mathcal{O}$ (*O grande*) mette in relazione la dimensione dei dati con il tempo medio di esecuzione;
2. Ω mette in relazione la dimensione dei dati con il tempo di esecuzione al caso migliore;
3. Θ mette in relazione la dimensione dei dati con il tempo di esecuzione al caso pessimo.

4.2.1 Notazione $\mathcal{O}$

Si utilizza la notazione $\mathcal{O}$ quando si vuole rappresentare un'approssimazione per eccesso del comportamento dell'algoritmo nel caso medio. Solitamente è la relazione più significativa tra tutte quelle prese in esame

$\mathcal{O}$

Sia $f(n)$ una funzione definita su $\mathbb{N}$ (gli interi non negativi).

Si dice che $T(n) \in \mathcal{O}(f(n))$ se $\exists$ una costante moltiplicativa $c \geq 0$ ed un numero intero $n_0 \geq 0 \Rightarrow \forall n > n_0, T(n) \leq cf(n)$.

Ad esempio, $n^2 + 2n + 4 \in \mathcal{O}(n^2)$, dato che esiste una coppia di costanti (n_0, c) che soddisfano la condizione richiesta, in questo caso, 4 e 2 sono due possibili scelte per n_0 e c. La definizione più formalmente corretta è che **il valore asintotico** del tempo medio di esecuzione dell'algoritmo è dato da n^2.

Le principali classi di complessità computazionale, espresse in notazione $\mathcal{O}$:

- $\mathcal{O}(1)$ indica un algoritmo il cui tempo di esecuzione è costante, ossia non dipende in alcun modo dall'input;
- $\mathcal{O}(\log n)$ indica un andamento logaritmico del tempo di esecuzione;
- $\mathcal{O}(n)$ indica un andamento lineare del tempo di esecuzione in funzione dell'input;
- $\mathcal{O}(n \log n)$ indica un andamento "poco più che lineare" del tempo di esecuzione, detto anche **linearitmico**;
- $\mathcal{O}(n^2)$ indica un andamento quadratico del tempo di esecuzione;
- $\mathcal{O}(n^3)$ indica un andamento cubico del tempo di esecuzione;
- $\mathcal{O}(2^n)$ indica un andamento esponenziale del tempo di esecuzione;
- $\mathcal{O}(n!)$ indica un andamento fattoriale del tempo di esecuzione;

4.2.2 Notazione Ω

La notazione Ω rappresenta un'approssimazione del tempo di esecuzione del programma nel caso ottimo, ossia quello in cui la soluzione viene raggiunta nel minor numero di passi possibile.

Ω

Sia $f(n)$ una funzione definita su $\mathbb{N}$ (gli interi non negativi).

Si dice che $T(n) \in \Omega(f(n))$ se $\exists$ una costante moltiplicativa $c \geq 0$ ed un numero intero $n_0 \geq 0$ tali che, $\forall n > n_0, T(n) \geq cf(n)$.

In altre parole, se un algoritmo è $\Omega(f(n))$, esso richiede non meno di $f(n)$ passi per essere risolto e quindi non si può pensare di ipotizzare una soluzione che appartenga a una classe di complessità migliore. Questa notazione rappresenta, quindi, anche il limite **inferiore** al tempo di esecuzione.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
n = 10	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec
n = 30	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	18min	10^{25}aa
n = 50	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	11min	36aa	$\gg$
n = 100	$<$ 1sec	$<$ 1sec	$<$ 1sec	$<$ 1sec	12.892aa	10^{17}aa	$\gg$
n = 1.000	$<$ 1sec	$<$ 1sec	1sec	18min	$\gg$	$\gg$	$\gg$
n = 10.000	$<$ 1sec	$<$ 1sec	2min	12gg	$\gg$	$\gg$	$\gg$
n = 100.000	$<$ 1sec	2sec	3ore	32aa	$\gg$	$\gg$	$\gg$
n = 1.000.000	1sec	20sec	12gg	31.710aa	$\gg$	$\gg$	$\gg$

Tabella 4.1: Una comparazione di come algoritmi appartenenti a diverse classi di complessità computazionale possano portare a tempi di esecuzione enormemente diversi tra loro; il simbolo $\gg$ indica un tempo di esecuzione superiore a 10^{25} anni, ricordando che l'età attuale dell'universo è all'incirca 10^{10} anni.

4.2.3 Notazione Θ

In modo simile ma opposto, la notazione Θ rappresenta un'approssimazione del tempo di esecuzione del programma nel caso pessimo, ossia quello in cui la soluzione viene raggiunta nel maggior numero di passi possibile.

❖ Θ ❖

Sia $f(n)$ una funzione definita su $\mathbb{N}$ (gli interi non negativi).

Si dice che $T(n) \in \Theta(f(n))$ se $T(n) \in \mathcal{O}(f(n))$ e $T(n) \in \Omega(f(n))$, ossia $T(n)$ cresce allo stesso andamento di $f(n)$.

In altre parole, un algoritmo è $\Theta(f(n))$ se non vi sono significative variazioni di prestazioni tra il caso migliore e il caso peggiore. Con questa notazione si rappresenta anche il limite **superiore** al tempo di esecuzione, ovvero il limite oltre il quale un algoritmo non deve mai andare nella soluzione del problema proposto.

4.2.4 Quale informazione è più utile?

Ciascuna delle tre notazioni contiene al suo interno importanti indizi sulla bontà dell'algoritmo e può fornire delle indicazioni su come utilizzare gli algoritmi stessi. La notazione Ω indica qual è il comportamento nel caso migliore e può essere utile nel caso in cui sia possibile effettuare del *pre-processing* sui dati, in modo da predisporli per il caso ottimo, oppure nel caso in cui si abbia la certezza che i dati presentino già una particolare struttura. Quando si ha la certezza teorica, tra l'altro, che la soluzione migliore ha una certa complessità

computazionale, e si è sviluppato un algoritmo che ha quella complessità, si può essere certi che **non è possibile far di meglio**.

La notazione Θ costituisce una stima del **tempo massimo di esecuzione** previsto dall'algoritmo.

La notazione $\mathcal{O}$ è quella che realmente indica il tempo medio di esecuzione ed è la notazione **largamente più utilizzata**, soprattutto quando non è possibile effettuare assunzioni sui dati. Poiché, tuttavia, non è realmente possibile effettuare delle misurazioni statistiche realmente esaustive, la notazione $\mathcal{O}$ va spesso a coincidere con la notazione Θ (una sorta di approssimazione per eccesso del caso medio). In sostanza, nella pratica, non potendo controllare quasi mai la disposizione dei dati con cui si effettua la computazione, la bontà di un algoritmo è misurata dalla notazione $\mathcal{O}$.

4.3 Complessità computazionale e linguaggio C

4.3.1 Complessità dei costrutti

Assegnamento

Per valutare la complessità computazionale di un algoritmo è necessario attribuire un *peso* a ciascun passo, che ne rappresenta il tempo di esecuzione. Utilizziamo esempi scritti in linguaggio C, anche se le considerazioni sviluppate sarebbero le medesime in qualunque modo (cioè qualunque linguaggio di programmazione utilizzassimo) noi descrivessimo l'algoritmo.

Un assegnamento singolo del tipo

```
x = 5;
```

coinvolge una scrittura in memoria (ed eventualmente una o più letture, se a sinistra dell'operatore sono presente una o più variabili da valutare). In generale, il tempo di esecuzione dell'operazione di assegnamento prescinde dalla grandezza dell'input: scrivere `x = 10` o `x = 10000` non cambia il tempo necessario ad eseguire l'operazione, quindi,la complessità computazionale dell'assegnamento è **costante**, che, riscritto in funzione della dimensione dell'input, si definisce come $\mathcal{O}(1)$.

Espressioni aritmetiche

La valutazione di una espressione aritmetica non dipende, ovviamente, dalla dimensione dei dati coinvolti: effettuare la valutazione di $5 + 10$ o quella di $5000 + 10000$ richiede la medesima quantità di tempo. Inoltre, anche se all'atto pratico può esistere una differenza nella valutazione di espressioni differenti (ad esempio, la moltiplicazione richiede una quantità di tempo superiore all'addizione per essere valutata), ai fini del calcolo globale si considera la valutazione come un'operazione con complessità costante $\mathcal{O}(1)$.

Un'istruzione del tipo

```
x = y*5;
```

in cui sono presenti sia la valutazione di una espressione che un assegnamento, continua ad avere complessità costante.

Espressioni relazionali

Le considerazioni fatte per le espressioni aritmetiche valgono anche per la valutazione di una espressione relazionale che coinvolge costanti e variabili e che non coinvolge funzioni. Un confronto del tipo

```
a<b
```

viene infatti valutato in tempo costante, in quanto esso non è influenzato dalla dimensione dei dati confrontati. Per estensione, espressioni relazionali concatenate da operatori logici booleani hanno anch'esse complessità costante $\mathcal{O}(1)$.

Costrutto di selezione

Si consideri un'istruzione della forma

```
if (condizione)
   BLOCCO1
else
   BLOCCO2
```

La valutazione della `condizione` la consideriamo avente complessità $\mathcal{O}(1)$ (è un'espressione relazionale), assumiamo poi che **blocco1** abbia una complessità $\mathcal{O}(f(n))$ e che **blocco2** abbia complessità $\mathcal{O}(g(n))$. L'intera istruzione ha una complessità data da

$$\mathcal{O}(1) + \max(\mathcal{O}(f(n)), \mathcal{O}(g(n)))$$

Infatti, poiché a priori non è possibile sapere quale dei due blocchi verrà eseguito, è necessario considerare la complessità computazionale massima (cioè il caso **pessimo**).

Costrutto iterativo enumerativo

Si consideri un'istruzione della forma

```
for(i=0; i < g(n); i ++)
   BLOCCO
```

Ciascuna delle tre componenti del ciclo `for` può essere considerata come eseguibile in tempo costante $\mathcal{O}(1)$ mentre sul **blocco** invece non è possibile fare assunzioni e ancora una volta dobbiamo assegnargli una complessità generica

$\mathcal{O}(f(n))$. Aggiungiamo un'istruzione di "salto" (di complessità costante) per tornare all'inizio del ciclo, il numero di iterazioni dipende direttamente dalla dimensione dei dati in gioco (ad esempio, la dimensione di un array), per cui **blocco** viene eseguito per un numero di volte che è determinato dalla complessità del ciclo $g(n)$. La formula finale della complessità è quindi data da

$$\mathcal{O}(1) + \sum_{g(n)} (\mathcal{O}(1) + \mathcal{O}(f(n)) + \mathcal{O}(1) + \mathcal{O}(1))$$

Costrutto iterativo a controllo logico

Si consideri un'istruzione della forma

```
while(a < b)
   BLOCCO
```

Supponiamo che, come nel caso precedente, il numero di iterazioni dipenda dalla dimensione del problema $g(n)$, allora la valutazione della condizione è costante, come è costante anche la complessità del salto a inizio ciclo e la complessità totale dell'istruzione è quindi data da

$$\sum_{g(n)} (\mathcal{O}(1) + \mathcal{O}(f(n)) + \mathcal{O}(1))$$

Stesse considerazioni (e conclusioni) valgono per il costrutto alternativo `do···while`. Questa definizione è applicabile soltanto se il numero di iterazioni è legato alla dimensione del problema, altrimenti è necessario stimare il numero di iterazioni in altro modo, non decidibile a priori in maniera standard, ma da valutare caso per caso.

4.3.2 Regole di semplificazione

Transitività

Si può utilizzare la regola di transitività per semplificare le espressioni $\mathcal{O}$

Transitività

Sia $f(n) \in \mathcal{O}(g(n))$ e $g(n) \in \mathcal{O}(h(n)) \Rightarrow f(n) \in \mathcal{O}(h(n))$

Alcuni esempi di applicazione della regola:

1. Si supponga $T(n) \in \mathcal{O}(n^2 - 1)$: poiché $n^2 - 1 \in n^2$ (si consideri, ad esempio, $n_0 = 0$, $c = 1$), allora $T(n) \in \mathcal{O}(n^2)$;

2. Si supponga $T(n) \in \mathcal{O}(n-1)$: poiché $n-1 \in n$ (si consideri, ad esempio, $n_0 = 0$, $c = 1$), allora $T(n) \in \mathcal{O}(n)$;

3. Si supponga $T(n) \in \mathcal{O}(n^2 - n)$: poiché $n^2 - n \in n^2$ (si consideri, ad esempio, $n_0 = 0$, $c = 1$), allora $T(n) \in \mathcal{O}(n^2)$.

Alla luce di questi casi, si può concludere che, ad esempio, se $T(n)$ è $\mathcal{O}(n)$, allora essa è anche $\mathcal{O}(n^2)$, $\mathcal{O}(n^3)$ e così via. Come si può decidere, quindi, quale debba essere la **vera** classe di complessità di un certo costrutto? La conclusione è più semplice di quanto si possa pensare: nel calcolo della complessità computazionale, è **fondamentale** determinare l'espressione $\mathcal{O}$ che sia il più stringente possibile, per cui, ad esempio, nel caso in esame si assegna il costrutto alla classe $\mathcal{O}(n)$ poiché è quella che meglio approssima il tempo di esecuzione in quanto, tra tutte quelle che possono essere scelta, è la più piccola.

Da questo è importante capire come sia essenziale avere un'idea chiara di quella che è la gerarchia delle classi di complessità computazionale per poter decidere quale scegliere.

Somma

Si supponga di avere un programma composto dalla sequenza di due blocchi con tempi di esecuzione

$$T_1(n) \in \mathcal{O}(f(n)) \wedge T_2(n) \in \mathcal{O}(g(n))$$

Si può dire che il tempo totale di esecuzione sia dato dalla somma dei tempi dei due costrutti

$$T(n) = T_1(n) + T_2(n) \in \mathcal{O}(f(n)) + \mathcal{O}(g(n)).$$

❖ **Somma** ❖
Supponendo che $g(n)$ sia $\mathcal{O}(f(n))$, allora $T(n) \in \mathcal{O}(f(n))$

Questa regola porta a rivedere e semplificare il modo in cui vengono valutate alcune istruzioni del linguaggio C. Abbiamo infatti visto che possiamo calcolare la complessità dell'istruzione `if` come:

$$\mathcal{O}(1) + max(\mathcal{O}(f(n)), \mathcal{O}(g(n)))$$

Si supponga, per ipotesi, che $\mathcal{O}(f(n))$ sia il massimo tra le due. Si può facilmente dimostrare che $1 \in \mathcal{O}(f(n))$: basta determinare n_0 e c tali che $cf(n) \geq 1$ e quindi l'espressione $\mathcal{O}(1) + \mathcal{O}(f(n))$ si semplifica in $\mathcal{O}(f(n)$. Allo stesso modo, se il massimo è $\mathcal{O}(g(n))$, l'intera espressione si semplifica in $\mathcal{O}(g(n))$. In conclusione, la complessità dell'istruzione `if-else` è data da:

$$max(\mathcal{O}(f(n)), \mathcal{O}(g(n)))$$

La complessità dell'istruzione `for` che abbiamo calcolato è:

$$\mathcal{O}(1) + \sum_{g(n)} (\mathcal{O}(1) + \mathcal{O}(f(n)) + \mathcal{O}(1) + \mathcal{O}(1))$$

Le espressioni $\mathcal{O}(1)$ possono essere semplificate tra loro, e la complessità può essere riscritta come:

$$\mathcal{O}(1) + \sum_{g(n)} (\mathcal{O}(1) + \mathcal{O}(f(n)))$$

che a sua volta diventa

$$\mathcal{O}(1) + \sum_{g(n)} (\mathcal{O}(f(n)))$$

Si consideri la sola sommatoria:

$$\sum_{g(n)} (\mathcal{O}(f(n)))$$

Essa equivale a:

$$g(n) * \mathcal{O}(f(n))$$

che a sua volta è

$$\mathcal{O}(g(n) * f(n))$$

Di conseguenza, la complessità dell'istruzione `for` è:

$$\mathcal{O}(1) + \mathcal{O}(g(n) * f(n)) \in \mathcal{O}(g(n) * f(n))$$

Seguendo lo stesso procedimento la complessità dell'istruzione `while`, che abbiamo calcolata come:

$$\sum_{g(n)} (\mathcal{O}(1) + \mathcal{O}(f(n)) + \mathcal{O}(1))$$

può essere semplificata sino ad ottenere la complessità:

$$\mathcal{O}(g(n) * f(n))$$

Possiamo dire empiricamente che le istruzioni più complesse **assorbono** nel calcolo della complessità quelle più semplici, poiché sono, di fatto, quelle che più significativamente determinano il tempo di esecuzione.

4.3.3 Esempi

Somma di n numeri

Torniamo adesso a valutare la complessità computazionale dei due algoritmi presentati inizialmente per il calcolo della somma di n numeri per cui eravamo giunti alla conclusione che il secondo metodo fosse, probabilmente, più efficiente del primo.

```
int i, n, sum=0;
...
for (i=1; i <= n; i++)
   sum += i;
```

Questo algoritmo ha complessità $\mathcal{O}(n)$ poiché l'istruzione con complessità costante `sum += i;` è ripetuta per `n` volte.

```
int n, sum=0;
...
sum = (n * (n+1))/2;
```

Questo algoritmo ha invece complessità $\mathcal{O}(1)$ poiché vi è una istruzione con complessità costante, eseguita una sola volta.

La verifica formale della complessità ci porta alla stessa conclusione che avevamo supposto intuitivamente.

Stampa di tutti gli elementi di un vettore

```
int voti[N];
int i;
...
for (int i = 0; i < N; i ++)
   printf("%d ", voti[i]);
```

In questo caso la complessità totale del costrutto è data dalla complessità dell'operazione `printf` moltiplicata per la complessità del costrutto iterativo enumerativo `for`. Ipotizzando che la complessità di `printf` sia $\mathcal{O}(g(n))$ si ottiene che la complessità globale dell'intero costrutto sia

$$\mathcal{O}(N * g(n))$$

Inversione di un vettore

```
int voti[N];
int i, j, tmp;
...
for (int i = 0, j = N-1; i < N/2; i ++, j = j-1) {
   tmp = voti[i];
   voti[i] = voti[j];
   voti[j] = tmp;
}
```

In questo spezzone di codice invertiamo le posizioni dei valori all'interno di un array, utilizzando due indici per scorrerlo nelle due opposte direzioni. La complessità totale del costrutto è data dalla complessità delle operazioni effettuate all'interno del ciclo moltiplicata per la complessità del costrutto iterativo enumerativo `for`. Poiché la complessità di ogni assegnamento è $\mathcal{O}(1)$ e, per la regola della somma, l'intera complessità delle istruzioni del ciclo è $\mathcal{O}(1)$ si ottiene che la complessità globale dell'intero costrutto sia

$$\mathcal{O}(N)$$

Stampa di tutti gli elementi di una matrice

```
int voti[N][M];
int i, j;
...
for (i = 0; i < N; i ++) {
   for (j = 0; j < M; j ++)
      printf("%d ", voti[i][j]);
   printf("\ n");
}
```

Supponiamo adesso per semplicità che la complessità dell'istruzione `printf` sia $\mathcal{O}(1)$. Abbiamo due cicli annidati uno dentro l'altro, quindi la complessità totale del costrutto è data dalla complessità del ciclo esterno $\mathcal{O}(N)$ moltiplicata per la complessità delle istruzioni eseguite all'interno del ciclo. Ma all'interno del ciclo eseguiamo un altro ciclo con complessità $\mathcal{O}(M)$, quindi la complessità totale del costrutto è

$$\mathcal{O}(M * N)$$

Appendice A

The Development of the C Language

Questo articolo è stato scritto da Dennis Ritchie nel 1993 per riassumere l'iter di sviluppo del linguaggio C.

A.1 Abstract

The C programming language was devised in the early 1970s as a system implementation language for the nascent Unix operating system. Derived from the typeless language BCPL, it evolved a type structure; created on a tiny machine as a tool to improve a meager programming environment, it has become one of the dominant languages of today. This paper studies its evolution.

A.2 Introduction

This paper[1] is about the development of the C programming language, the influences on it, and the conditions under which it was created. For the sake of brevity, I omit full descriptions of C itself, its parent B [Johnson 73] and its grandparent BCPL [Richards 79], and instead concentrate on characteristic elements of each language and how they evolved.

[1] This article was presented at Second History of Programming Languages conference, Cambridge, Mass., April, 1993. It was then collected in the conference proceedings: *History of Programming Languages-II* ed. Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. ACM Press (New York) and Addison-Wesley (Reading, Mass), 1996; ISBN 0-201-89502-1.

C came into being in the years 1969-1973, in parallel with the early development of the Unix operating system; the most creative period occurred during 1972. Another spate of changes peaked between 1977 and 1979, when portability of the Unix system was being demonstrated. In the middle of this second period, the first widely available description of the language appeared: *The C Programming Language*, often called the 'white book' or 'K&R' [Kernighan 78]. Finally, in the middle 1980s, the language was officially standardized by the ANSI X3J11 committee, which made further changes. Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, the language was almost exclusively associated with Unix; more recently, its use has spread much more widely, and today it is among the languages most commonly used throughout the computer industry.

A.3 History: the setting

The late 1960s were a turbulent era for computer systems research at Bell Telephone Laboratories [Ritchie 78] [Ritchie 84]. The company was pulling out of the Multics project [Organick 75], which had started as a joint venture of MIT, General Electric, and Bell Labs; by 1969, Bell Labs management, and even the researchers, came to believe that the promises of Multics could be fulfilled only too late and too expensively. Even before the GE-645 Multics machine was removed from the premises, an informal group, led primarily by Ken Thompson, had begun investigating alternatives.

Thompson wanted to create a comfortable computing environment constructed according to his own design, using whatever means were available. His plans, it is evident in retrospect, incorporated many of the innovative aspects of Multics, including an explicit notion of a process as a locus of control, a tree-structured file system, a command interpreter as user-level program, simple representation of text files, and generalized access to devices. They excluded others, such as unified access to memory and to files. At the start, moreover, he and the rest of us deferred another pioneering (though not original) element of Multics, namely writing almost exclusively in a higher-level language. PL/I, the implementation language of Multics, was not much to our tastes, but we were also using other languages, including BCPL, and we regretted losing the advantages of writing programs in a language above the level of assembler, such as ease of writing and clarity of understanding. At the time we did not put much weight on portability; interest in this arose later.

Thompson was faced with a hardware environment cramped and spartan even for the time: the DEC PDP-7 on which he started in 1968 was a machine with 8K 18-bit words of memory and no software useful to him. While wanting to use a higher-level language, he wrote the original Unix system in PDP-7 assembler. At the start, he did not even program on the PDP-7 itself, but instead used a set of macros for the GEMAP assembler on a GE-635 machine. A postprocessor generated a paper tape readable by the PDP-7.

These tapes were carried from the GE machine to the PDP-7 for testing until a primitive Unix kernel, an editor, an assembler, a simple shell (command interpreter), and a few utilities (like the Unix *rm, cat, cp* commands) were completed. After this point, the operating system was self-supporting: programs could be written and tested without resort to paper tape, and development continued on the PDP-7 itself.

Thompson's PDP-7 assembler outdid even DEC's in simplicity; it evaluated expressions and emitted the corresponding bits. There were no libraries, no loader or link editor: the entire source of a program was presented to the assembler, and the output file – with a fixed name – that emerged was directly executable. (This name, *a.out*, explains a bit of Unix etymology; it is the output of the assembler. Even after the system gained a linker and a means of specifying another name explicitly, it was retained as the default executable result of a compilation.)

Not long after Unix first ran on the PDP-7, in 1969, Doug McIlroy created the new system's first higher-level language: an implementation of McClure's TMG [McClure 65]. TMG is a language for writing compilers (more generally, TransMoGrifiers) in a top-down, recursive-descent style that combines context-free syntax notation with procedural elements. McIlroy and Bob Morris had used TMG to write the early PL/I compiler for Multics.

Challenged by McIlroy's feat in reproducing TMG, Thompson decided that Unix – possibly it had not even been named yet – needed a system programming language. After a rapidly scuttled attempt at Fortran, he created instead a language of his own, which he called B. B can be thought of as C without types; more accurately, it is BCPL squeezed into 8K bytes of memory and filtered through Thompson's brain. Its name most probably represents a contraction of BCPL, though an alternate theory holds that it derives from Bon [Thompson 69], an unrelated language created by Thompson during the Multics days. Bon in turn was named either after his wife Bonnie, or (according to an encyclopedia quotation in its manual), after a religion whose rituals involve the murmuring of magic formulas.

A.4 Origins: the languages

BCPL was designed by Martin Richards in the mid-1960s while he was visiting MIT, and was used during the early 1970s for several interesting projects, among them the OS6 operating system at Oxford [Stoy 72], and parts of the seminal Alto work at Xerox PARC [Thacker 79]. We became familiar with it because the MIT CTSS system [Corbato 62] on which Richards worked was used for Multics development. The original BCPL compiler was transported both to Multics and to the GE-635 GECOS system by Rudd Canaday and others at Bell Labs [Canaday 69]; during the final throes of Multics's life at Bell Labs and immediately after, it was the language of choice among the group of people who would later become involved with Unix.

BCPL, B, and C all fit firmly in the traditional procedural family typified by Fortran and Algol 60. They are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are 'close to the machine' in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system. With less success, they also use library procedures to specify interesting control constructs such as coroutines and procedure closures. At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved.

BCPL, B and C differ syntactically in many details, but broadly they are similar. Programs consist of a sequence of global declarations and function (procedure) declarations. Procedures can be nested in BCPL, but may not refer to non-static objects defined in containing procedures. B and C avoid this restriction by imposing a more severe one: no nested procedures at all. Each of the languages (except for earliest versions of B) recognizes separate compilation, and provides a means for including text from named files.

Several syntactic and lexical mechanisms of BCPL are more elegant and regular than those of B and C. For example, BCPL's procedure and data declarations have a more uniform structure, and it supplies a more complete set of looping constructs. Although BCPL programs are notionally supplied from an undelimited stream of characters, clever rules allow most semicolons to be elided after statements that end on a line boundary. B and C omit this convenience, and end most statements with semicolons. In spite of the differences, most of the statements and operators of BCPL map directly into corresponding B and C.

Some of the structural differences between BCPL and B stemmed from limitations on intermediate memory. For example, BCPL declarations may take the form

```
let P1 be command
and P2 be command
and P3 be command
 ...
```

where the program text represented by the commands contains whole procedures. The subdeclarations connected by `and` occur simultaneously, so the name `P3` is known inside procedure `P1`. Similarly, BCPL can package a group of declarations and statements into an expression that yields a value, for example

```
E1 := valof (declarations;commands;resultis E2) + 1
```

The BCPL compiler readily handled such constructs by storing and analyzing a parsed representation of the entire program in memory before producing

output. Storage limitations on the B compiler demanded a one-pass technique in which output was generated as soon as possible, and the syntactic redesign that made this possible was carried forward into C.

Certain less pleasant aspects of BCPL owed to its own technological problems and were consciously avoided in the design of B. For example, BCPL uses a 'global vector' mechanism for communicating between separately compiled programs. In this scheme, the programmer explicitly associates the name of each externally visible procedure and data object with a numeric offset in the global vector; the linkage is accomplished in the compiled code by using these numeric offsets. B evaded this inconvenience initially by insisting that the entire program be presented all at once to the compiler. Later implementations of B, and all those of C, use a conventional linker to resolve external names occurring in files compiled separately, instead of placing the burden of assigning offsets on the programmer.

Other fiddles in the transition from BCPL to B were introduced as a matter of taste, and some remain controversial, for example the decision to use the single character = for assignment instead of `:=`. Similarly, B uses `/**/` to enclose comments, where BCPL uses `//`, to ignore text up to the end of the line. The legacy of PL/I is evident here. (C++ has resurrected the BCPL comment convention.) Fortran influenced the syntax of declarations: B declarations begin with a specifier like `auto` or `static`, followed by a list of names, and C not only followed this style but ornamented it by placing its type keywords at the start of declarations.

Not every difference between the BCPL language documented in Richards's book [Richards 79] and B was deliberate; we started from an earlier version of BCPL [Richards 67]. For example, the `endcase` that escapes from a BCPL `switchon` statement was not present in the language when we learned it in the 1960s, and so the overloading of the `break` keyword to escape from the B and C `switch` statement owes to divergent evolution rather than conscious change.

In contrast to the pervasive syntax variation that occurred during the creation of B, the core semantic content of BCPL – its type structure and expression evaluation rules – remained intact. Both languages are typeless, or rather have a single data type, the 'word,' or 'cell,' a fixed-length bit pattern. Memory in these languages consists of a linear array of such cells, and the meaning of the contents of a cell depends on the operation applied. The + operator, for example, simply adds its operands using the machine's integer add instruction, and the other arithmetic operations are equally unconscious of the actual meaning of their operands. Because memory is a linear array, it is possible to interpret the value in a cell as an index in this array, and BCPL supplies an operator for this purpose. In the original language it was spelled `rv`, and later `!`, while B uses the unary `*`. Thus, if `p` is a cell containing the index of (or address of, or pointer to) another cell, `*p` refers to the contents of the pointed-to cell, either as a value in an expression or as the target of an assignment.

Because pointers in BCPL and B are merely integer indices in the memory array, arithmetic on them is meaningful: if `p` is the address of a cell, then `p+1` is the address of the next cell. This convention is the basis for the semantics of arrays in both languages. When in BCPL one writes

```
let V = vec 10
```

or in B,

```
auto V[10];
```

the effect is the same: a cell named `V` is allocated, then another group of 10 contiguous cells is set aside, and the memory index of the first of these is placed into `V`. By a general rule, in B the expression

```
*(V+i)
```

adds `V` and `i`, and refers to the `i`-th location after `V`. Both BCPL and B each add special notation to sweeten such array accesses; in B an equivalent expression is

```
V[i]
```

and in BCPL

```
V!i
```

This approach to arrays was unusual even at the time; C would later assimilate it in an even less conventional way.

None of BCPL, B, or C supports character data strongly in the language; each treats strings much like vectors of integers and supplements general rules by a few conventions. In both BCPL and B a string literal denotes the address of a static area initialized with the characters of the string, packed into cells. In BCPL, the first packed byte contains the number of characters in the string; in B, there is no count and strings are terminated by a special character, which B spelled '`*e`'. This change was made partially to avoid the limitation on the length of a string caused by holding the count in an 8- or 9-bit slot, and partly because maintaining the count seemed, in our experience, less convenient than using a terminator.

Individual characters in a BCPL string were usually manipulated by spreading the string out into another array, one character per cell, and then repacking it later; B provided corresponding routines, but people more often used other library functions that accessed or replaced individual characters in a string.

A.5 More History

After the TMG version of B was working, Thompson rewrote B in itself (a bootstrapping step). During development, he continually struggled against memory limitations: each language addition inflated the compiler so it could barely fit, but each rewrite taking advantage of the feature reduced its size. For example, B introduced generalized assignment operators, using `x=+y` to add `y` to `x`. The notation came from Algol 68 [Wijngaarden 75] via McIlroy, who had incorporated it into his version of TMG. (In B and early C, the operator was spelled =+ instead of += ; this mistake, repaired in 1976, was induced by a seductively easy way of handling the first form in B's lexical analyzer.)

Thompson went a step further by inventing the ++ and −− operators, which increment or decrement; their prefix or postfix position determines whether the alteration occurs before or after noting the value of the operand. They were not in the earliest versions of B, but appeared along the way. People often guess that they were created to use the auto-increment and auto-decrement address modes provided by the DEC PDP-11 on which C and Unix first became popular. This is historically impossible, since there was no PDP-11 when B was developed. The PDP-7, however, did have a few 'auto-increment' memory cells, with the property that an indirect memory reference through them incremented the cell. This feature probably suggested such operators to Thompson; the generalization to make them both prefix and postfix was his own. Indeed, the auto-increment cells were not used directly in implementation of the operators, and a stronger motivation for the innovation was probably his observation that the translation of `++x` was smaller than that of `x=x+1`.

The B compiler on the PDP-7 did not generate machine instructions, but instead 'threaded code' [Bell 72], an interpretive scheme in which the compiler's output consists of a sequence of addresses of code fragments that perform the elementary operations. The operations typically – in particular for B – act on a simple stack machine.

On the PDP-7 Unix system, only a few things were written in B except B itself, because the machine was too small and too slow to do more than experiment; rewriting the operating system and the utilities wholly into B was too expensive a step to seem feasible. At some point Thompson relieved the address-space crunch by offering a 'virtual B' compiler that allowed the interpreted program to occupy more than 8K bytes by paging the code and data within the interpreter, but it was too slow to be practical for the common utilities. Still, some utilities written in B appeared, including an early version of the variable-precision calculator *dc* familiar to Unix users [McIlroy 79]. The most ambitious enterprise I undertook was a genuine cross-compiler that translated B to GE-635 machine instructions, not threaded code. It was a small *tour de force*: a full B compiler, written in its own language and generating code for a 36-bit mainframe, that ran on an 18-bit machine with 4K words of user address space. This project was possible only because of the simplicity of the B language and its run-time

system.

Although we entertained occasional thoughts about implementing one of the major languages of the time like Fortran, PL/I, or Algol 68, such a project seemed hopelessly large for our resources: much simpler and smaller tools were called for. All these languages influenced our work, but it was more fun to do things on our own.

By 1970, the Unix project had shown enough promise that we were able to acquire the new DEC PDP-11. The processor was among the first of its line delivered by DEC, and three months passed before its disk arrived. Making B programs run on it using the threaded technique required only writing the code fragments for the operators, and a simple assembler which I coded in B; soon, *dc* became the first interesting program to be tested, before any operating system, on our PDP-11. Almost as rapidly, still waiting for the disk, Thompson recoded the Unix kernel and some basic commands in PDP-11 assembly language. Of the 24K bytes of memory on the machine, the earliest PDP-11 Unix system used 12K bytes for the operating system, a tiny space for user programs, and the remainder as a RAM disk. This version was only for testing, not for real work; the machine marked time by enumerating closed knight's tours on chess boards of various sizes. Once its disk appeared, we quickly migrated to it after transliterating assembly-language commands to the PDP-11 dialect, and porting those already in B.

By 1971, our miniature computer center was beginning to have users. We all wanted to create interesting software more easily. Using assembler was dreary enough that B, despite its performance problems, had been supplemented by a small library of useful service routines and was being used for more and more new programs. Among the more notable results of this period was Steve Johnson's first version of the *yacc* parser-generator [Johnson 79a].

A.6 The Problems of B

The machines on which we first used BCPL and then B were word-addressed, and these languages' single data type, the 'cell,' comfortably equated with the hardware machine word. The advent of the PDP-11 exposed several inadequacies of B's semantic model. First, its character-handling mechanisms, inherited with few changes from BCPL, were clumsy: using library procedures to spread packed strings into individual cells and then repack, or to access and replace individual characters, began to feel awkward, even silly, on a byte-oriented machine.

Second, although the original PDP-11 did not provide for floating-point arithmetic, the manufacturer promised that it would soon be available. Floating-point operations had been added to BCPL in our Multics and GCOS compilers by defining special operators, but the mechanism was possible only because on the relevant machines, a single word was large enough to contain a floating-point number; this was not true on the 16-bit PDP-11.

Finally, the B and BCPL model implied overhead in dealing with pointers: the language rules, by defining a pointer as an index in an array of words, forced pointers to be represented as word indices. Each pointer reference generated a run-time scale conversion from the pointer to the byte address expected by the hardware.

For all these reasons, it seemed that a typing scheme was necessary to cope with characters and byte addressing, and to prepare for the coming floating-point hardware. Other issues, particularly type safety and interface checking, did not seem as important then as they became later.

Aside from the problems with the language itself, the B compiler's threaded-code technique yielded programs so much slower than their assembly-language counterparts that we discounted the possibility of recoding the operating system or its central utilities in B.

In 1971 I began to extend the B language by adding a character type and also rewrote its compiler to generate PDP-11 machine instructions instead of threaded code. Thus the transition from B to C was contemporaneous with the creation of a compiler capable of producing programs fast and small enough to compete with assembly language. I called the slightly-extended language NB, for 'new B.'

A.7 Embryonic C

NB existed so briefly that no full description of it was written. It supplied the types `int` and `char`, arrays of them, and pointers to them, declared in a style typified by

```
int i, j;
char c, d;
int iarray[10];
int ipointer[];
char carray[10];
char cpointer[];
```

The semantics of arrays remained exactly as in B and BCPL: the declarations of `iarray` and `carray` create cells dynamically initialized with a value pointing to the first of a sequence of 10 integers and characters respectively. The declarations for `ipointer` and `cpointer` omit the size, to assert that no storage should be allocated automatically. Within procedures, the language's interpretation of the pointers was identical to that of the array variables: a pointer declaration created a cell differing from an array declaration only in that the programmer was expected to assign a referent, instead of letting the compiler allocate the space and initialize the cell.

Values stored in the cells bound to array and pointer names were the machine addresses, measured in bytes, of the corresponding storage area. Therefore,

indirection through a pointer implied no run-time overhead to scale the pointer from word to byte offset. On the other hand, the machine code for array subscripting and pointer arithmetic now depended on the type of the array or the pointer: to compute `iarray[i]` or `ipointer+i` implied scaling the addend `i` by the size of the object referred to.

These semantics represented an easy transition from B, and I experimented with them for some months. Problems became evident when I tried to extend the type notation, especially to add structured (record) types. Structures, it seemed, should map in an intuitive way onto memory in the machine, but in a structure containing an array, there was no good place to stash the pointer containing the base of the array, nor any convenient way to arrange that it be initialized. For example, the directory entries of early Unix systems might be described in C as

```
struct {
    int number;
    char name[14];
};
```

I wanted the structure not merely to characterize an abstract object but also to describe a collection of bits that might be read from a directory. Where could the compiler hide the pointer to `name` that the semantics demanded? Even if structures were thought of more abstractly, and the space for pointers could be hidden somehow, how could I handle the technical problem of properly initializing these pointers when allocating a complicated object, perhaps one that specified structures containing arrays containing structures to arbitrary depth?

The solution constituted the crucial jump in the evolutionary chain between typeless BCPL and typed C. It eliminated the materialization of the pointer in storage, and instead caused the creation of the pointer when the array name is mentioned in an expression. The rule, which survives in today's C, is that values of array type are converted, when they appear in expressions, into pointers to the first of the objects making up the array.

This invention enabled most existing B code to continue to work, despite the underlying shift in the language's semantics. The few programs that assigned new values to an array name to adjust its origin – possible in B and BCPL, meaningless in C – were easily repaired. More important, the new language retained a coherent and workable (if unusual) explanation of the semantics of arrays, while opening the way to a more comprehensive type structure.

The second innovation that most clearly distinguishes C from its predecessors is this fuller type structure and especially its expression in the syntax of declarations. NB offered the basic types `int` and `char`, together with arrays of them, and pointers to them, but no further ways of composition. Generalization was required: given an object of any type, it should be possible to describe a

new object that gathers several into an array, yields it from a function, or is a pointer to it.

For each object of such a composed type, there was already a way to mention the underlying object: index the array, call the function, use the indirection operator on the pointer. Analogical reasoning led to a declaration syntax for names mirroring that of the expression syntax in which the names typically appear. Thus,

```
int i, *pi, **ppi;
```

declare an integer, a pointer to an integer, a pointer to a pointer to an integer. The syntax of these declarations reflects the observation that `i`, `*pi`, and `**ppi` all yield an `int` type when used in an expression. Similarly,

```
int f(), *f(), (*f)();
```

declare a function returning an integer, a function returning a pointer to an integer, a pointer to a function returning an integer;

```
int *api[10], (*pai)[10];
```

declare an array of pointers to integers, and a pointer to an array of integers. In all these cases the declaration of a variable resembles its usage in an expression whose type is the one named at the head of the declaration.

The scheme of type composition adopted by C owes considerable debt to Algol 68, although it did not, perhaps, emerge in a form that Algol's adherents would approve of. The central notion I captured from Algol was a type structure based on atomic types (including structures), composed into arrays, pointers (references), and functions (procedures). Algol 68's concept of unions and casts also had an influence that appeared later.

After creating the type system, the associated syntax, and the compiler for the new language, I felt that it deserved a new name; NB seemed insufficiently distinctive. I decided to follow the single-letter style and called it C, leaving open the question whether the name represented a progression through the alphabet or through the letters in BCPL.

A.8 Neonatal C

Rapid changes continued after the language had been named, for example the introduction of the and `||` operators. In BCPL and B, the evaluation of expressions depends on context: within `if` and other conditional statements that compare an expression's value with zero, these languages place a special interpretation on the `and` (`&`) and `or` (`|`) operators. In ordinary contexts, they operate bitwise, but in the B statement

```
if (e1 \& e2) ...
```

the compiler must evaluate `e1` and if it is non-zero, evaluate `e2`, and if it too is non-zero, elaborate the statement dependent on the `if`. The requirement descends recursively on and `|` operators within `e1` and `e2`. The short-circuit semantics of the Boolean operators in such 'truth-value' context seemed desirable, but the overloading of the operators was difficult to explain and use. At the suggestion of Alan Snyder, I introduced the and `||` operators to make the mechanism more explicit.

Their tardy introduction explains an infelicity of C's precedence rules. In B one writes

```
if (a==b \& c) ...
```

to check whether `a` equals `b` and `c` is non-zero; in such a conditional expression it is better that have lower precedence than ==. In converting from B to C, one wants to replace by in such a statement; to make the conversion less painful, we decided to keep the precedence of the operator the same relative to ==, and merely split the precedence of slightly from . Today, it seems that it would have been preferable to move the relative precedences of and ==, and thereby simplify a common C idiom: to test a masked value against another value, one must write

```
if ((a\&mask) == b) ...
```

where the inner parentheses are required but easily forgotten.

Many other changes occurred around 1972-3, but the most important was the introduction of the preprocessor, partly at the urging of Alan Snyder [Snyder 74], but also in recognition of the utility of the the file-inclusion mechanisms available in BCPL and PL/I. Its original version was exceedingly simple, and provided only included files and simple string replacements: `include` and `define` of parameterless macros. Soon thereafter, it was extended, mostly by Mike Lesk and then by John Reiser, to incorporate macros with arguments and conditional compilation. The preprocessor was originally considered an optional adjunct to the language itself. Indeed, for some years, it was not even invoked unless the source program contained a special signal at its beginning. This attitude persisted, and explains both the incomplete integration of the syntax of the preprocessor with the rest of the language and the imprecision of its description in early reference manuals.

A.9 Portability

By early 1973, the essentials of modern C were complete. The language and compiler were strong enough to permit us to rewrite the Unix kernel for the

PDP-11 in C during the summer of that year. (Thompson had made a brief attempt to produce a system coded in an early version of C – before structures – in 1972, but gave up the effort.) Also during this period, the compiler was retargeted to other nearby machines, particularly the Honeywell 635 and IBM 360/370; because the language could not live in isolation, the prototypes for the modern libraries were developed. In particular, Lesk wrote a 'portable I/O package' [Lesk 72] that was later reworked to become the C 'standard I/O' routines. In 1978 Brian Kernighan and I published *The C Programming Language* [Kernighan 78]. Although it did not describe some additions that soon became common, this book served as the language reference until a formal standard was adopted more than ten years later. Although we worked closely together on this book, there was a clear division of labor: Kernighan wrote almost all the expository material, while I was responsible for the appendix containing the reference manual and the chapter on interfacing with the Unix system.

During 1973-1980, the language grew a bit: the type structure gained unsigned, long, union, and enumeration types, and structures became nearly first-class objects (lacking only a notation for literals). Equally important developments appeared in its environment and the accompanying technology. Writing the Unix kernel in C had given us enough confidence in the language's usefulness and efficiency that we began to recode the system's utilities and tools as well, and then to move the most interesting among them to the other platforms. As described in [Johnson 78a], we discovered that the hardest problems in propagating Unix tools lay not in the interaction of the C language with new hardware, but in adapting to the existing software of other operating systems. Thus Steve Johnson began to work on *pcc*, a C compiler intended to be easy to retarget to new machines [Johnson 78b], while he, Thompson, and I began to move the Unix system itself to the Interdata 8/32 computer.

The language changes during this period, especially around 1977, were largely focused on considerations of portability and type safety, in an effort to cope with the problems we foresaw and observed in moving a considerable body of code to the new Interdata platform. C at that time still manifested strong signs of its typeless origins. Pointers, for example, were barely distinguished from integral memory indices in early language manuals or extant code; the similarity of the arithmetic properties of character pointers and unsigned integers made it hard to resist the temptation to identify them. The `unsigned` types were added to make unsigned arithmetic available without confusing it with pointer manipulation. Similarly, the early language condoned assignments between integers and pointers, but this practice began to be discouraged; a notation for type conversions (called 'casts' from the example of Algol 68) was invented to specify type conversions more explicitly. Beguiled by the example of PL/I, early C did not tie structure pointers firmly to the structures they pointed to, and permitted programmers to write `pointer->member` almost without regard to the type of `pointer`; such an expression was taken uncritically as a reference to a region of memory designated by the pointer, while the member name specified only an offset and a type.

Although the first edition of K&R described most of the rules that brought C's type structure to its present form, many programs written in the older, more relaxed style persisted, and so did compilers that tolerated it. To encourage people to pay more attention to the official language rules, to detect legal but suspicious constructions, and to help find interface mismatches undetectable with simple mechanisms for separate compilation, Steve Johnson adapted his *pcc* compiler to produce *lint* [Johnson 79b], which scanned a set of files and remarked on dubious constructions.

A.10 Growth in Usage

The success of our portability experiment on the Interdata 8/32 soon led to another by Tom London and John Reiser on the DEC VAX 11/780. This machine became much more popular than the Interdata, and Unix and the C language began to spread rapidly, both within AT&T and outside. Although by the middle 1970s Unix was in use by a variety of projects within the Bell System as well as a small group of research-oriented industrial, academic, and government organizations outside our company, its real growth began only after portability had been achieved. Of particular note were the System III and System V versions of the system from the emerging Computer Systems division of AT&T, based on work by the company's development and research groups, and the BSD series of releases by the University of California at Berkeley that derived from research organizations in Bell Laboratories.

During the 1980s the use of the C language spread widely, and compilers became available on nearly every machine architecture and operating system; in particular it became popular as a programming tool for personal computers, both for manufacturers of commercial software for these machines, and for end-users interested in programming. At the start of the decade, nearly every compiler was based on Johnson's *pcc*; by 1985 there were many independently-produced compiler products.

A.11 Standardization

By 1982 it was clear that C needed formal standardization. The best approximation to a standard, the first edition of K&R, no longer described the language in actual use; in particular, it mentioned neither the `void` or `enum` types. While it foreshadowed the newer approach to structures, only after it was published did the language support assigning them, passing them to and from functions, and associating the names of members firmly with the structure or union containing them. Although compilers distributed by AT&T incorporated these changes, and most of the purveyors of compilers not based on *pcc* quickly picked up them up, there remained no complete, authoritative description of the language.

The first edition of K&R was also insufficiently precise on many details of the language, and it became increasingly impractical to regard *pcc* as a 'reference compiler;' it did not perfectly embody even the language described by K&R, let alone subsequent extensions. Finally, the incipient use of C in projects subject to commercial and government contract meant that the imprimatur of an official standard was important. Thus (at the urging of M. D. McIlroy), ANSI established the X3J11 committee under the direction of CBEMA in the summer of 1983, with the goal of producing a C standard. X3J11 produced its report [ANSI 89] at the end of 1989, and subsequently this standard was accepted by ISO as ISO/IEC 9899-1990.

From the beginning, the X3J11 committee took a cautious, conservative view of language extensions. Much to my satisfaction, they took seriously their goal: 'to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments.' [ANSI 89] The committee realized that mere promulgation of a standard does not make the world change.

X3J11 introduced only one genuinely important change to the language itself: it incorporated the types of formal arguments in the type signature of a function, using syntax borrowed from C++ [Stroustrup 86]. In the old style, external functions were declared like this:

```
double sin();
```

which says only that `sin` is a function returning a `double` (that is, double-precision floating-point) value. In the new style, this better rendered

```
double sin(double);
```

to make the argument type explicit and thus encourage better type checking and appropriate conversion. Even this addition, though it produced a noticeably better language, caused difficulties. The committee justifiably felt that simply outlawing 'old-style' function definitions and declarations was not feasible, yet also agreed that the new forms were better. The inevitable compromise was as good as it could have been, though the language definition is complicated by permitting both forms, and writers of portable software must contend with compilers not yet brought up to standard.

X3J11 also introduced a host of smaller additions and adjustments, for example, the type qualifiers `const` and `volatile`, and slightly different type promotion rules. Nevertheless, the standardization process did not change the character of the language. In particular, the C standard did not attempt to specify formally the language semantics, and so there can be dispute over fine points; nevertheless, it successfully accounted for changes in usage since the original description, and is sufficiently precise to base implementations on it.

Thus the core C language escaped nearly unscathed from the standardization process, and the Standard emerged more as a better, careful codification than a new invention. More important changes took place in the language's surroundings: the preprocessor and the library. The preprocessor performs macro substitution, using conventions distinct from the rest of the language. Its interaction with the compiler had never been well-described, and X3J11 attempted to remedy the situation. The result is noticeably better than the explanation in the first edition of K&R; besides being more comprehensive, it provides operations, like token concatenation, previously available only by accidents of implementation.

X3J11 correctly believed that a full and careful description of a standard C library was as important as its work on the language itself. The C language itself does not provide for input-output or any other interaction with the outside world, and thus depends on a set of standard procedures. At the time of publication of K&R, C was thought of mainly as the system programming language of Unix; although we provided examples of library routines intended to be readily transportable to other operating systems, underlying support from Unix was implicitly understood. Thus, the X3J11 committee spent much of its time designing and documenting a set of library routines required to be available in all conforming implementations.

By the rules of the standards process, the current activity of the X3J11 committee is confined to issuing interpretations on the existing standard. However, an informal group originally convened by Rex Jaeschke as NCEG (Numerical C Extensions Group) has been officially accepted as subgroup X3J11.1, and they continue to consider extensions to C. As the name implies, many of these possible extensions are intended to make the language more suitable for numerical use: for example, multi-dimensional arrays whose bounds are dynamically determined, incorporation of facilities for dealing with IEEE arithmetic, and making the language more effective on machines with vector or other advanced architectural features. Not all the possible extensions are specifically numerical; they include a notation for structure literals.

A.12 Successors

C and even B have several direct descendants, though they do not rival Pascal in generating progeny. One side branch developed early. When Steve Johnson visited the University of Waterloo on sabbatical in 1972, he brought B with him. It became popular on the Honeywell machines there, and later spawned Eh and Zed (the Canadian answers to 'what follows B?'). When Johnson returned to Bell Labs in 1973, he was disconcerted to find that the language whose seeds he brought to Canada had evolved back home; even his own *yacc* program had been rewritten in C, by Alan Snyder.

More recent descendants of C proper include Concurrent C [Gehani 89], Objective C [Cox 86], C* [Thinking 90], and especially C++ [Stroustrup 86].

The language is also widely used as an intermediate representation (essentially, as a portable assembly language) for a wide variety of compilers, both for direct descendents like C++, and independent languages like Modula 3 [Nelson 91] and Eiffel [Meyer 88].

A.13 Critique

Two ideas are most characteristic of C among languages of its class: the relationship between arrays and pointers, and the way in which declaration syntax mimics expression syntax. They are also among its most frequently criticized features, and often serve as stumbling blocks to the beginner. In both cases, historical accidents or mistakes have exacerbated their difficulty. The most important of these has been the tolerance of C compilers to errors in type. As should be clear from the history above, C evolved from typeless languages. It did not suddenly appear to its earliest users and developers as an entirely new language with its own rules; instead we continually had to adapt existing programs as the language developed, and make allowance for an existing body of code. (Later, the ANSI X3J11 committee standardizing C would face the same problem.)

Compilers in 1977, and even well after, did not complain about usages such as assigning between integers and pointers or using objects of the wrong type to refer to structure members. Although the language definition presented in the first edition of K&R was reasonably (though not completely) coherent in its treatment of type rules, that book admitted that existing compilers didn't enforce them. Moreover, some rules designed to ease early transitions contributed to later confusion. For example, the empty square brackets in the function declaration

```
int f(a) int a[]; { ... }
```

are a living fossil, a remnant of NB's way of declaring a pointer; `a` is, in this special case only, interpreted in C as a pointer. The notation survived in part for the sake of compatibility, in part under the rationalization that it would allow programmers to communicate to their readers an intent to pass `f` a pointer generated from an array, rather than a reference to a single integer. Unfortunately, it serves as much to confuse the learner as to alert the reader.

In K&R C, supplying arguments of the proper type to a function call was the responsibility of the programmer, and the extant compilers did not check for type agreement. The failure of the original language to include argument types in the type signature of a function was a significant weakness, indeed the one that required the X3J11 committee's boldest and most painful innovation to repair. The early design is explained (if not justified) by my avoidance of technological problems, especially cross-checking between separately-compiled source files, and my incomplete assimilation of the implications of moving

between an untyped to a typed language. The *lint* program, mentioned above, tried to alleviate the problem: among its other functions, *lint* checks the consistency and coherency of a whole program by scanning a set of source files, comparing the types of function arguments used in calls with those in their definitions.

An accident of syntax contributed to the perceived complexity of the language. The indirection operator, spelled `*` in C, is syntactically a unary prefix operator, just as in BCPL and B. This works well in simple expressions, but in more complex cases, parentheses are required to direct the parsing. For example, to distinguish indirection through the value returned by a function from calling a function designated by a pointer, one writes `*fp()` and `(*pf)()` respectively. The style used in expressions carries through to declarations, so the names might be declared

```
int *fp();
int (*pf)();
```

In more ornate but still realistic cases, things become worse:

```
int *(*pfp)();
```

is a pointer to a function returning a pointer to an integer. There are two effects occurring. Most important, C has a relatively rich set of ways of describing types (compared, say, with Pascal). Declarations in languages as expressive as C – Algol 68, for example – describe objects equally hard to understand, simply because the objects themselves are complex. A second effect owes to details of the syntax. Declarations in C must be read in an 'inside-out' style that many find difficult to grasp [Anderson 80]. Sethi [Sethi 81] observed that many of the nested declarations and expressions would become simpler if the indirection operator had been taken as a postfix operator instead of prefix, but by then it was too late to change.

In spite of its difficulties, I believe that the C's approach to declarations remains plausible, and am comfortable with it; it is a useful unifying principle. The other characteristic feature of C, its treatment of arrays, is more suspect on practical grounds, though it also has real virtues. Although the relationship between pointers and arrays is unusual, it can be learned. Moreover, the language shows considerable power to describe important concepts, for example, vectors whose length varies at run time, with only a few basic rules and conventions. In particular, character strings are handled by the same mechanisms as any other array, plus the convention that a null character terminates a string. It is interesting to compare C's approach with that of two nearly contemporaneous languages, Algol 68 and Pascal [Jensen 74]. Arrays in Algol 68 either have fixed bounds, or are 'flexible:' considerable mechanism is required both in the language definition, and in compilers, to accommodate flexible arrays (and not all compilers fully implement them.) Original Pascal had only fixed-sized

arrays and strings, and this proved confining [Kernighan 81]. Later, this was partially fixed, though the resulting language is not yet universally available.

C treats strings as arrays of characters conventionally terminated by a marker. Aside from one special rule about initialization by string literals, the semantics of strings are fully subsumed by more general rules governing all arrays, and as a result the language is simpler to describe and to translate than one incorporating the string as a unique data type. Some costs accrue from its approach: certain string operations are more expensive than in other designs because application code or a library routine must occasionally search for the end of a string, because few built-in operations are available, and because the burden of storage management for strings falls more heavily on the user. Nevertheless, C's approach to strings works well.

On the other hand, C's treatment of arrays in general (not just strings) has unfortunate implications both for optimization and for future extensions. The prevalence of pointers in C programs, whether those declared explicitly or arising from arrays, means that optimizers must be cautious, and must use careful dataflow techniques to achieve good results. Sophisticated compilers can understand what most pointers can possibly change, but some important usages remain difficult to analyze. For example, functions with pointer arguments derived from arrays are hard to compile into efficient code on vector machines, because it is seldom possible to determine that one argument pointer does not overlap data also referred to by another argument, or accessible externally. More fundamentally, the definition of C so specifically describes the semantics of arrays that changes or extensions treating arrays as more primitive objects, and permitting operations on them as wholes, become hard to fit into the existing language. Even extensions to permit the declaration and use of multidimensional arrays whose size is determined dynamically are not entirely straightforward [MacDonald 89] [Ritchie 90], although they would make it much easier to write numerical libraries in C. Thus, C covers the most important uses of strings and arrays arising in practice by a uniform and simple mechanism, but leaves problems for highly efficient implementations and for extensions.

Many smaller infelicities exist in the language and its description besides those discussed above, of course. There are also general criticisms to be lodged that transcend detailed points. Chief among these is that the language and its generally-expected environment provide little help for writing very large systems. The naming structure provides only two main levels, 'external' (visible everywhere) and 'internal' (within a single procedure). An intermediate level of visibility (within a single file of data and procedures) is weakly tied to the language definition. Thus, there is little direct support for modularization, and project designers are forced to create their own conventions.

Similarly, C itself provides two durations of storage: 'automatic' objects that exist while control resides in or below a procedure, and 'static,' existing throughout execution of a program. Off-stack, dynamically-allocated storage is

provided only by a library routine and the burden of managing it is placed on the programmer: C is hostile to automatic garbage collection.

A.14 Whence Success?

C has become successful to an extent far surpassing any early expectations. What qualities contributed to its widespread use?

Doubtless the success of Unix itself was the most important factor; it made the language available to hundreds of thousands of people. Conversely, of course, Unix's use of C and its consequent portability to a wide variety of machines was important in the system's success. But the language's invasion of other environments suggests more fundamental merits.

Despite some aspects mysterious to the beginner and occasionally even to the adept, C remains a simple and small language, translatable with simple and small compilers. Its types and operations are well-grounded in those provided by real machines, and for people used to how computers work, learning the idioms for generating time- and space-efficient programs is not difficult. At the same time the language is sufficiently abstracted from machine details that program portability can be achieved.

Equally important, C and its central library support always remained in touch with a real environment. It was not designed in isolation to prove a point, or to serve as an example, but as a tool to write programs that did useful things; it was always meant to interact with a larger operating system, and was regarded as a tool to build larger tools. A parsimonious, pragmatic approach influenced the things that went into C: it covers the essential needs of many programmers, but does not try to supply too much.

Finally, despite the changes that it has undergone since its first published description, which was admittedly informal and incomplete, the actual C language as seen by millions of users using many different compilers has remained remarkably stable and unified compared to those of similarly widespread currency, for example Pascal and Fortran. There are differing dialects of C – most noticeably, those described by the older K&R and the newer Standard C – but on the whole, C has remained freer of proprietary extensions than other languages. Perhaps the most significant extensions are the 'far' and 'near' pointer qualifications intended to deal with peculiarities of some Intel processors. Although C was not originally designed with portability as a prime goal, it succeeded in expressing programs, even including operating systems, on machines ranging from the smallest personal computers through the mightiest supercomputers.

C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.

A.15 Acknowledgments

It is worth summarizing compactly the roles of the direct contributors to today's C language. Ken Thompson created the B language in 1969-70; it was derived directly from Martin Richards's BCPL. Dennis Ritchie turned B into C during 1971-73, keeping most of B's syntax while adding types and many other changes, and writing the first compiler. Ritchie, Alan Snyder, Steven C. Johnson, Michael Lesk, and Thompson contributed language ideas during 1972-1977, and Johnson's portable compiler remains widely used. During this period, the collection of library routines grew considerably, thanks to these people and many others at Bell Laboratories. In 1978, Brian Kernighan and Ritchie wrote the book that became the language definition for several years. Beginning in 1983, the ANSI X3J11 committee standardized the language. Especially notable in keeping its efforts on track were its officers Jim Brodie, Tom Plum, and P. J. Plauger, and the successive draft redactors, Larry Rosler and Dave Prosser.

I thank Brian Kernighan, Doug McIlroy, Dave Prosser, Peter Nelson, Rob Pike, Ken Thompson, and HOPL's referees for advice in the preparation of this paper.