Dispense delle lezioni di Fondamenti di Programmazione

Introduzione al linguaggio C

Matteo Fraschini

23/06/2026

Indice

Prefazione

Caro studente, questo documento è per te. So bene che seguire le lezioni a volte può essere faticoso e che, anche quando l'interesse è alto, ci sono mille motivi che rendono difficile tornare a casa e mettersi subito a studiare. Qui troverai i **concetti fondamentali** che abbiamo trattato in aula, sintetizzati per offrirti una guida chiara su ciò che ritengo davvero essenziale.

Oggi, nell'era del "**vibe coding**" in cui basta descrivere un programma in linguaggio naturale perché un LLM lo generi, studiare la programmazione può sembrare un esercizio inutile. Non è così. L'IA ha reso invisibile il livello più basso dell'informatica, ma quel livello esiste ancora (ci sono ancora la memoria da gestire, i puntatori, i compilatori e le sequenze di bit). Se non capisci cosa succede sotto l'astrazione, non avrai mai gli strumenti per accorgerti quando l'IA produce un codice errato o inefficiente. Il ruolo del programmatore si sta spostando verso l'alto, verso la progettazione e la supervisione critica. Per questo serve ancora una comprensione profonda.

Questo testo è un "**lavoro in corso**". Ovviamente non sostituisce i libri di testo e, soprattutto, non sarà sufficiente se non dedicherai del tempo a sperimentare direttamente. Imparare a programmare non significa semplicemente memorizzare la sintassi di un linguaggio, richiede creatività, pensiero logico e una buona dose di determinazione.

Una caratteristica inevitabile di queste dispense, e della programmazione in generale, è che alcuni argomenti non possono essere introdotti in ordine sequenziale. Accadrà di usare uno strumento prima di averne compreso appieno il funzionamento, o di ricevere una spiegazione parziale che si completerà solo più avanti, quando avrai le conoscenze necessarie per affrontarla in modo rigoroso. Quando ti troverai in questa situazione, non preoccuparti. Prendi nota del dubbio e vai avanti.

Per ogni altra informazione in merito al corso, alla sua organizzazione e alla modalità di verifica dell'apprendimento fai riferimento alla mia pagina personale.

1 Introduzione

È possibile che, in questo preciso momento, il tuo livello sia **0**. Significa solo che, come altri tuoi colleghi, non hai mai scritto una riga di codice prima d'ora. **Non preoccuparti, non è affatto un problema**. Questa dispensa è nata proprio per studenti come te e non richiede alcun prerequisito specifico. Il numero 0 è fondamentale in informatica e lo incontrerai molto spesso in queste pagine. Presto il tuo livello cambierà, crescerà e si arricchirà di nuove competenze, modificando l'offset rispetto al punto di partenza. Ricordati di questo "Livello 0" quando arriveremo a studiare gli array, capirai che iniziare a contare da zero può essere incredibilmente rilevante!

Il percorso è organizzato in livelli progressivi e si affronteranno i seguenti argomenti:

- Introduzione al C
- La struttura di un programma
- Le variabili
- Tipo di dati semplici
- Strutture di controllo decisionali
- Strutture di controllo iterative
- Array
- Puntatori
- Funzioni
- Stringhe
- Strutture
- Oggetti dinamici
- File

I Livelli sono concepiti per essere trasversali e il loro scopo non è isolare la teoria, ma mostrare come i singoli argomenti si intrecciano nella pratica.

Viviamo in un mondo in cui la tecnologia è ovunque. Scegliere di capire cosa c'è sotto il cofano non ti darà solo un vantaggio tecnico, ma migliorerà drasticamente le tue competenze di logica e di problem-solving, formandoti una struttura mentale utile in qualsiasi ambito. Programmare è un'attività gratificante, ti mette da subito nelle condizioni di fare qualcosa di nuovo, trasformando un'idea astratta in qualcosa di reale e funzionante.

2 Livello 0

In questo capitolo muoveremo i primi passi essenziali, sia pratici che teorici, dalla configurazione dell'ambiente di sviluppo alla comprensione del ciclo di compilazione, fino alla scrittura e all'analisi riga per riga del tuo primo programma.

2.1 Configurare l'ambiente di sviluppo

Tutto ciò di cui hai bisogno per iniziare è configurare il tuo ambiente di sviluppo (quello che in gergo chiamiamo IDE). Esistono diverse soluzioni per farlo, e la scelta finale dipenderà in parte dal tuo Sistema Operativo (Windows, macOS o Linux), quindi scegli pure quella che preferisci o con cui ti trovi più a tuo agio.

Ecco le due strade principali che ti consiglio:

- **CLion** (scelta consigliata): come studente di UniCa, puoi richiedere una licenza annuale gratuita (rinnovabile) per tutti i prodotti JetBrains, incluso CLion, uno dei migliori ambienti di sviluppo per il C. Ti basta registrarti con la tua mail istituzionale ed è praticamente pronto all'uso, senza configurazioni complesse. Link per la licenza studenti.
- **Visual Studio Code** (scelta leggera e personalizzabile): la migliore alternativa "leggera" è VS Code. È gratuito, open-source e consuma pochissime risorse. Tieni conto, però, che richiede un minimo di configurazione iniziale in quanto dovrai installare l'estensione per il C/C++ e assicurarti di avere un compilatore (come GCC o Clang) installato sul tuo computer.

Il mio consiglio è di non perdere giorni a cercare l'ambiente di sviluppo "perfetto". Scegli quello che riesci a installare più facilmente sul tuo computer e inizia a scrivere codice.

Se vuoi capire meglio cosa succede senza delegare nulla a interfacce grafiche complesse, puoi scrivere il codice su un editor di testo (come Notepad++, Sublime Text, o persino il Blocco Note) e usare il terminale (la riga di comando) per compilarlo ed eseguirlo.

2.2 Editor di testo e terminale

In questo ultimo caso ecco come procedere:

- Apri il tuo editor, scrivi il programma e salvalo con l'estensione .c (ad esempio, main.c).
- Apri il terminale e spostati nella cartella in cui hai salvato il file usando il comando cd.
- Digiti il comando per trasformare il testo in un file binario eseguibile.
- Lanci il programma appena generato.

Su macOS: i Mac utilizzano il compilatore clang (che si attiva anche digitando il comando storico gcc). Se il terminale dovesse dirti che il comando non esiste, ti basterà installare il tool a riga di comando digitando `xcode-select --install`.

Una volta installato, i comandi nel Terminale sono:

- `gcc -Wall main.c -o il_mio_programma` per compilare
- `./il_mio_programma` per eseguire

Su Windows: puoi usare il prompt dei comandi (CMD). Di base Windows non include un compilatore C. Il modo più pulito per ottenerlo è installare una distribuzione di GCC per Windows (come MinGW-w64, installabile facilmente tramite MSYS2) e assicurarsi di inserire il percorso del compilatore nelle variabili d'ambiente del sistema.

Una volta configurato GCC, i comandi sono del tutto analoghi:

- `gcc -Wall main.c -o il_mio_programma` per compilare
- `il_mio_programma.exe` per eseguire

In Windows, per eseguire il programma, il prefisso `.\` è obbligatorio se utilizzi la PowerShell.

> **Il flag -Wall**
>
> Negli esempi di compilazione è presente `-Wall` (Warning All). Questo comando dice al compilatore di segnalare qualsiasi riga di codice che, pur non bloccando la creazione del programma, risulta potenzialmente problematica.

2.3 Linguaggi di programmazione

Per capire dove si colloca il C, dobbiamo fare un piccolo passo indietro. La storia dell'informatica è legata ad una progressiva astrazione. Esistono tre categorie di linguaggi:

- Linguaggio Macchina: linguaggio che il computer comprende nativamente, è composto esclusivamente da sequenze di bit (0 e 1), tanto elementare per l'hardware quanto impossibile da gestire per un essere umano.
- Linguaggio Assembly: rappresenta il primo livello di astrazione e sostituisce gli 0 e 1 con parole chiave (come ADD o MOV), rendendo il codice leggibile dall'uomo, pur rimanendo legato all'architettura della macchina.
- Linguaggi di Alto Livello: linguaggi vicini alla logica umana e indipendenti dall'hardware. A seconda di come vengono tradotti per la macchina, si dividono in *compilati* (il codice viene tradotto tutto insieme prima dell'esecuzione) o *interpretati* (tradotti riga per riga mentre il programma è in esecuzione).

Il C è un linguaggio di alto livello di tipo compilato. Quando scriviamo un programma in C, il nostro testo non può essere eseguito direttamente ma deve attraversare un flusso di lavoro diviso in 6 fasi fondamentali:

- Editing: il programmatore scrive il codice sorgente (es. main.c) utilizzando un editor di testo.
- Preprocessing: il preprocessore analizza il codice sorgente ed esegue tutte le direttive che iniziano con il simbolo `#` (come `#include` e `#define`), include i file richiesti, effettua le sostituzioni e produce il codice sorgente che verrà effettivamente compilato.

- Compilazione: un software chiamato compilatore traduce il codice sorgente in linguaggio macchina, generando il cosiddetto codice oggetto (un file intermedio, non ancora pronto per l'uso).
- Linking: il linker unisce il codice oggetto appena creato con le funzioni delle librerie di sistema. Il risultato di questa unione è il file eseguibile finale (il .exe su Windows).
- Loading: quando lanci il programma, il sistema operativo copia il file eseguibile dal disco fisso alla memoria centrale (RAM) del computer.
- Esecuzione: la CPU legge le istruzioni presenti nella memoria centrale e le esegue una dopo l'altra.

Le ultime due (Loading ed Esecuzione) sono fasi runtime gestite dal Sistema Operativo.

2.4 La struttura di un programma in C

Ora che sei pronto, è arrivato il momento di fare una prova pratica con un programma semplicissimo. Tutto quello che devi fare è aprire un nuovo file nel tuo editor, assegnargli un nome e salvarlo con l'estensione .c.

Riscrivi nel tuo editor il codice che trovi qui sotto:

```c
//Il mio primo programma in C

#include <stdio.h>
int main(void) {
  printf("Hello world\n");
  return 0;
}
```

A questo punto, non ti resta che compilare ed eseguire il programma. Se tutto è configurato nel modo corretto, il codice girerà senza problemi e non dovresti ricevere alcun tipo di errore o di avviso (*warning*). Cosa significa questo all'atto pratico? Significa che vedrai comparire la scritta "Hello world" nel terminale (o nella console integrata del tuo IDE).

Questo è il più semplice programma in C che puoi realizzare.

> **E in Python?**
>
> Anche se non hai mai programmato prima penso sia intuitivo capire quanto questo programma in C sia più complesso di quello che potresti scrivere in Python.
>
> ```python
> # Il mio primo programma in Python
>
> print("Hello world")
> ```

Prima però di passare a qualcosa di più complesso e decisamente più interessante, fermiamoci un istante a guardare cosa hai appena scritto nel tuo file .c. Sono convinto che comprendere a fondo la struttura di base di un programma in C sia un passo fondamentale per il tuo percorso, ed è molto più facile farlo adesso quando il codice è ancora composto da pochissime righe.

Analizziamolo quindi riga per riga.

- I commenti (`// Il mio primo programma in C`). Se nel tuo codice inserisci una riga che inizia con due barre oblique (`//`), stai scrivendo un commento. Questa riga viene completamente ignorata dal compilatore. Tuttavia, usare i commenti all'interno dei tuoi script è fondamentale. Ti consiglio di inserirne il più possibile. Saranno di vitale importanza quando riaprirai il file a distanza di settimane o mesi per capire cosa avevi in mente. Oltre ai commenti su singola riga esiste una seconda forma, utile quando il testo si estende su più righe (tutto ciò che è racchiuso tra `/*` e `*/` viene ignorato dal compilatore).
- La direttiva `#include <stdio.h>`. Questa riga serve a includere informazioni contenute nella libreria stdio.h (Standard Input and Output), libreria che mette a disposizione, tra le altre cose, la funzione `printf` che abbiamo usato per stampare il testo a video. Più avanti scopriremo altre librerie utilissime per i nostri progetti. Per precisione, questa riga non include una "libreria" vera e propria, ma un header file (da cui l'estensione .h), un file di testo che contiene le dichiarazioni delle funzioni messe a disposizione dalla libreria. Il codice vero e proprio della funzione, invece, verrà unito al tuo programma in un secondo momento, durante la fase di linking che abbiamo descritto poco fa.
- La funzione principale: `int main(void)`. Ogni singolo programma scritto in C deve avere una funzione principale. Questa funzione si chiama `main`. L'esecuzione del tuo programma partirà da lì. Studieremo le funzioni nel dettaglio più avanti, per ora ti basta sapere che il `main` racchiude le istruzioni che il computer dovrà eseguire. La parola `int` prima del nome indica che questa funzione, alla fine del suo lavoro, deve restituire un numero intero. Il `main` (come ogni altra funzione) ha quindi un nome, un valore di ritorno, eventualmente un elenco di argomenti in ingresso tra le parentesi tonde e un corpo, ovvero lo spazio in cui risiede il codice vero e proprio, delimitato dalle parentesi graffe `{ }`. La parola `void` indica che la funzione `main` non accetta in questo caso alcun parametro (o argomento) in ingresso.
- La chiusura: `return 0;`. L'ultima istruzione del corpo del `main` è `return 0;`. Questo è il valore intero che la funzione restituisce per comunicare che tutto è andato a buon fine. Se ometti questa riga il programma funzionerà comunque, ma scoprirai presto che non si tratta di un aspetto estetico. Per la **sola funzione** `main`, lo standard del linguaggio (dal C99 in poi) stabilisce che, se ometti il `return`, il programma restituisce comunque 0 in automatico.

Una piccola nota: il simbolo `\n` in coda rappresenta un ritorno a capo e serve a far ripartire il prompt del terminale su una riga nuova. Ne riparleremo a breve.

> **i** Quale versione del C?
>
> Il C è un linguaggio standardizzato che si è evoluto nel tempo attraverso diverse versioni (C89, C99, C11, C17, C23). In questa dispensa facciamo riferimento al C moderno (dal C99 in poi), che è quello supportato senza configurazioni particolari da tutti i compilatori attuali.

3 Livello 1

Il presente capitolo introduce i costrutti fondamentali per la gestione dei dati nel linguaggio C. Vengono trattati in modo sistematico il concetto di variabile e la sua rappresentazione in memoria, le strategie per la definizione delle costanti, l'utilizzo degli operatori aritmetici e le caratteristiche dei tipi di dato semplici.

3.1 Le variabili

Ora sei pronto a fare un passo avanti e utilizzare le variabili. Per capire cosa siano, immaginale semplicemente come dei contenitori destinati a ospitare i dati che il tuo programma dovrà usare. Puoi fare riferimento a questo contenitore (un piccolo blocco della memoria RAM) usando un nome scelto da te. Sei libero di chiamare una variabile quasi come preferisci (ci sono solo pochissime regole sintattiche da rispettare), ma il consiglio è di scegliere sempre nomi significativi, che facciano capire immediatamente cosa c'è dentro.

In C prima di poter usare una variabile, devi obbligatoriamente dichiararla. La dichiarazione è un passaggio cruciale perché associa alla tua variabile un tipo. Quando assegni un tipo a una variabile, stai definendo due cose fondamentali:

- La dimensione del contenitore (quanti byte di memoria deve riservare a quella variabile).
- Le operazioni consentite (cosa puoi fare con quel dato).

Ricorda che la dimensione esatta in byte associata ad un tipo di dato non è fissa a priori, ma può dipendere dall'architettura hardware della macchina su cui viene eseguito il programma (ad esempio, se la CPU è a 32 o a 64 bit). Ecco un esempio di come si presenta una dichiarazione di variabile in C:

```c
int number; // variabile di tipo intero di nome è number
```

Con questa istruzione stai dichiarando una variabile chiamata `number` come un numero intero (`int`). In sostanza viene riservato un piccolo blocco di memoria RAM per questa variabile (quasi certamente 4 byte) e, da questo momento in poi potrai accedere al contenuto di quel blocco di RAM semplicemente usando il nome della variabile.

In Figura 3.1, gli effetti in RAM della dichiarazione della variabile `number`.

Il linguaggio C è *case sensitive*, significa che è sensibile alla differenza tra lettere maiuscole e lettere minuscole. I programmatori usano delle convenzioni standard per dare i nomi alle variabili. In C la più diffusa è lo *snake_case*, in cui le parole sono scritte in minuscolo e separate dal carattere underscore. In altri linguaggi è molto comune anche il *camelCase* (chiamato così perché le maiuscole in mezzo alla parola ricordano le gobbe di un cammello).

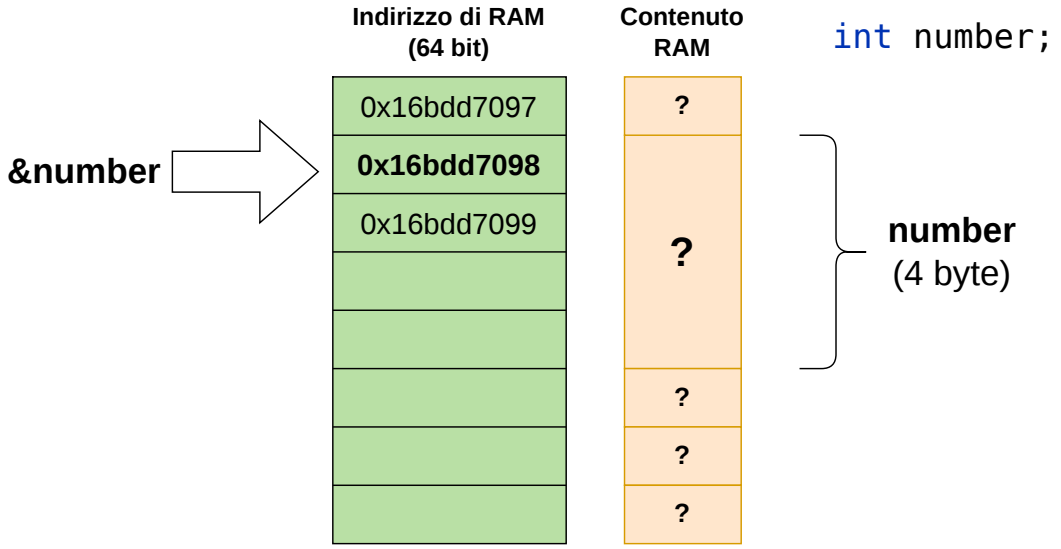

Figura 3.1: Schema dell'organizzazione in RAM della variabile `number`. Il `?` indica il fatto che il valore contenuto nei 4 byte non è noto.

3.1.1 Variabili e dimensione

Quanto è grande questo blocco di memoria? Nella maggior parte delle architetture moderne la risposta è 4 byte, ma puoi verificarlo tu stesso direttamente sul tuo computer con un piccolo test:

```c
#include <stdio.h>
int main(void) {
  int number; // variabile di tipo intero
  printf("\nUn intero occupa %zu byte", sizeof(number));
  // In alternativa:
  // printf("\nUn intero occupa %zu byte", sizeof(int));
  return 0;
}
```

In questo blocco di codice, `sizeof` ha il compito di determinare la dimensione fisica in byte occupata in memoria RAM dalla variabile `number` o, nell'alternativa commentata, direttamente dal tipo di dato `int`. Come puoi vedere, la funzione `printf` può essere usata anche per stampare il risultato di un'espressione. Per farlo, devi semplicemente indicare al C dove vuoi che il valore appaia all'interno della stringa di testo. Questo si fa inserendo uno specificatore di formato (`%zu` fa riferimento al tipo `size_t` intero senza segno definito dallo standard C per rappresentare la dimensione in byte di qualsiasi oggetto in memoria), dopodiché, ti basta indicare l'espressione o la variabile subito dopo la virgola.

3.1.2 Variabili e contenuto

Ma se `number` fa riferimento ad un blocco della RAM, qual è il suo contenuto? Proviamo a verificarlo.

```c
#include <stdio.h>
int main(void) {
    int number;
    printf("Contenuto della variabile number: %d", number);
    // %d è il segnaposto del valore contenuto nella variabile number
    return 0;
}
```

Se provi a compilare ed eseguire questo codice, potresti rimanere sorpreso. Il compilatore non si bloccherà e sul terminale potrebbe apparire un numero apparentemente senza senso (si genera quello che formalmente si chiama comportamento indefinito). Per capire cosa è successo, dobbiamo guardare di nuovo sotto il cofano. Quando hai scritto `int number;` è stata cercata nella RAM una porzione libera (da 4 byte) ed è stata usata per la variabile `number`. Quel numero che vedi a schermo non è altro che l'interpretazione come numero intero della sequenza di bit che già si trovavano in quella porzione di RAM. Questi valori "spazzatura" sono residui lasciati in quella zona di memoria dal tuo stesso programma.

Posso ovviamente accedere a quel blocco di RAM e modificarne il contenuto:

```c
#include <stdio.h>
int main(void) {
    int number = 111;
    printf("Contenuto della variabile number: %d", number);
    return 0;
}
```

L'istruzione `=`, nota come assegnazione, consente di sovrascrivere (e quindi perdere definitivamente) il contenuto precedente. Da questo momento la variabile `number` contiene il valore intero `111`, o meglio, il blocco di RAM di 4 byte che tu chiami `number` conterrà una sequenza di bit (32 bit) associabile a quel numero intero.

> **!** Importante
>
> È di fondamentale importanza non confondere il simbolo `=` con il simbolo `==`.
>
> - `=` (assegnamento): viene usato per inserire (e sovrascrivere) un valore dentro una variabile.
> - `==` (confronto): viene usato per verificare un'uguaglianza, cioè per controllare se due variabili o espressioni hanno lo stesso valore.
>
> Se usi per errore il simbolo `=` dove invece volevi usare `==`, il compilatore non segnalerà alcun errore di sintassi. Il programma verrà generato ed eseguito normalmente, ma si comporterà in modo completamente diverso da ciò che ti aspetti.

3.1.3 Variabili e indirizzo

È possibile conoscere l'indirizzo di RAM che ospita il valore associato ad una particolare variabile? Partiamo da questo frammento di codice:

```c
#include <stdio.h>
int main(void) {
    int x, y;
    x = 99;
    printf("\n%d",x);
    printf("\n%p",&x);
    x = x + 1;
    printf("\n%d",x);
    printf("\n%p",&x);
    y = x;
    printf("\n%d",y);
    printf("\n%p",&y);
    return 0;
}
```

Questo programma ci permette di "fotografare" la RAM in tre momenti diversi, svelando la differenza fondamentale tra cosa contiene una variabile e dove si trova fisicamente. Per farlo, introduciamo un nuovo strumento:

- L'operatore `&` (address-of) anteposto al nome di una variabile, non legge il suo valore, ma chiede di restituire l'indirizzo di memoria.

Lo specificatore `%p` indica alla funzione `printf` che vogliamo stampare un indirizzo di memoria, che verrà mostrato sullo schermo in formato esadecimale (una sequenza di numeri e lettere, ad esempio `0x16b87f098`).

Lo standard richiederebbe di convertire l'indirizzo al tipo generico `void *` prima di stamparlo, ovvero `printf("%p", (void *)&x);`. I compilatori comuni accettano anche la forma semplificata, ma è bene conoscere quella formalmente corretta.

Analizziamo cosa succede durante l'esecuzione. All'inizio dichiariamo due variabili, `x` e `y`. Il compilatore riserva due spazi distinti nella RAM. Quando scriviamo `x = 99;`, riempiamo il primo contenitore. Le prime due `printf` mostreranno 99 (il valore memorizzato) e 0x… (l'indirizzo fisico di x). L'istruzione `x = x + 1;` incrementa il valore di `x`, che diventa 100. Questa è un'operazione distruttiva e il vecchio 99 è perso per sempre. Le successive due `printf` ci mostrano che il valore è ora 100 e l'indirizzo di memoria è identico a quello di prima. Le variabili in C hanno un domicilio fisso. Nel corso del programma puoi cambiare il valore, ma il contenitore non si sposta mai dalla cella di memoria che gli è stata assegnata. L'assegnamento `y = x` crea una copia. Sappiamo che `x` in questo momento vale 100. Cosa succede nella RAM? Le ultime `printf` rivelano che la variabile `y` stampa il valore 100 e l'indirizzo di memoria di `y` è completamente diverso da quello di `x`. In sostanza, "il computer" è andato all'indirizzo di `x`, ha preso i 32 bit che componevano il numero 100, ed è andato a copiarli nell'indirizzo di `y`. Da questo momento in poi, `x` e `y` vivono vite separate. Se modifichi `y`, il valore di `x` non cambierà.

> **E in Python?**
>
> Il Python le cose sono molto diverse. A differenza del C, in Python una variabile è semplicemente un etichetta attaccata a un oggetto. In Python, le variabili non hanno un tipo, sono gli oggetti ad averlo. Questo cambio di filosofia spiega perché in Python non esiste la necessità di dichiarare le variabili e perché si ha la sensazione che una variabile possa cambiare tipo nello stesso programma.

> In Python è perfettamente legale scrivere `x = 1` e subito dopo `x = "ciao"`. A seguito della prima istruzione viene creato nella RAM un oggetto di tipo intero che contiene il valore 1 e poi l'etichetta `x` viene attaccata a quell'oggetto. A seguito della seconda istruzione viene creato un nuovo oggetto, questa volta di tipo stringa, contenente il testo "ciao" e a questo punto, viene attaccata al nuovo oggetto di tipo stringa. L'etichetta `x` è rimasta identica ma è cambiato l'oggetto a cui fa riferimento.

3.2 Le costanti

Fino a questo momento abbiamo parlato di variabili, il cui valore può cambiare nel tempo. Spesso sorge l'esigenza di utilizzare valori che devono rimanere immutabili per tutta la durata del programma. In C esistono due strategie diverse per definire una costante: la parola chiave `const` e la direttiva al preprocessore `#define`.

L'utilizzo di `const` qualifica una variabile "in sola lettura". Per esempio con `const float PiGreco = 3.14;` il valore viene assegnato in modo permanente e il compilatore impedirà qualsiasi tentativo di modifica. Dal punto di vista della memoria, si comportano come normali variabili in quanto hanno un tipo associato e rispettano le regole di visibilità (si dice che sono "scope-controlled") che vedremo in seguito.

La direttiva al preprocessore `#define` è diversa, non definisce una variabile, ma è un'istruzione rivolta al preprocessore (ovvero a quel componente che analizza il codice sorgente prima che inizi la compilazione vera e propria). Per esempio con `#define N 100` il preprocessore esegue un lavoro di "trova e sostituisci" su tutto il file di testo (ogni volta che incontra i simbolo N scrive al suo posto 100).

3.3 Operatori aritmetici

Il C mette a disposizione i classici operatori per effettuare calcoli numerici sulle variabili e sulle espressioni. Di seguito sono elencati gli operatori fondamentali:

- `+` Addizione: calcola la somma tra due operandi.
- `-` Sottrazione: calcola la differenza tra due operandi.
- `*` Moltiplicazione: calcola il prodotto tra due operandi.
- `/` Divisione: calcola il quoziente tra due operandi.
- `%` Modulo: calcola il resto della divisione intera.

Il comportamento dell'operatore di divisione `/` dipende dal tipo di dato degli operandi. Se entrambi gli operandi sono interi esegue una divisione intera troncando la parte decimale. Se almeno uno degli operandi è un numero a virgola mobile esegue una divisione reale.

L'operatore modulo `%` richiede invece operandi interi.

3.4 Tipi di dato

Torniamo ora al legame tra dichiarazioni e tipi. In C esistono pochi tipi fondamentali (chiamati anche tipi base). Più avanti vedremo che potrai definire dei nuovi tipi personalizzati, ma per adesso i mattoncini di partenza sono quattro:

- `int`: utilizzato per i numeri interi (es. 10, -5, 0).
- `float`: utilizzato per i numeri reali a singola precisione (numeri con la virgola, es. 3.14).
- `double`: utilizzato per i numeri reali a doppia precisione (più precisi e capienti dei float).
- `char`: utilizzato per i singoli caratteri di testo (es. 'A', 'b', '?').

La scelta di quale tipo assegnare a una variabile dipenderà esclusivamente dal contesto e dagli obiettivi del tuo programma. Ricorda che da questa decisione dipenderanno sempre due aspetti cruciali: le operazioni che potrai effettuare su quella variabile e la dimensione del blocco di memoria che il computer riserverà.

> **!** Importante
>
> A ciascun tipo di dato è associato un preciso intervallo di valori rappresentabili. Questo limite è una conseguenza della quantità di memoria fisica che il compilatore alloca per quel tipo, la quale determina il numero massimo di configurazioni binarie distinte che l'hardware può assumere. La dimensione effettiva e l'intervallo associato dipendono dall'architettura del processore.

Per fissare le idee, ecco dimensioni su un moderno PC a 64 bit:

Tipo	Dimensione tipica
`char`	1 byte
`int`	4 byte
`float`	4 byte
`double`	8 byte

Esistono inoltre dei qualificatori che permettono di modificare dimensione e intervallo dei tipi interi: `short` e `long` agiscono sulla dimensione (es. `short int`, `long int`), mentre `unsigned` rinuncia ai numeri negativi per raddoppiare il massimo valore rappresentabile. I valori esatti validi per la tua piattaforma sono definiti negli header di sistema `<limits.h>` (per gli interi) e `<float.h>` (per i tipi a virgola mobile).

4 Livello 2

Il presente capitolo analizza i meccanismi di interazione tra il programma e l'ambiente esterno attraverso i canali di input e output standard. Viene approfondito l'utilizzo della funzione `scanf()` per l'acquisizione di dati da tastiera, dettagliando il ruolo cruciale dell'operatore di indirizzo `&` nel contesto del passaggio dei parametri per valore e le dinamiche hardware legate alle violazioni di memoria del tipo Segmentation Fault.

4.1 Istruzioni di input e output

Un programma necessita di interagire con l'ambiente esterno per acquisire dati da elaborare e per mostrare i risultati delle elaborazioni. In C, questa interazione avviene principalmente attraverso i canali standard di comunicazione (standard I/O), gestiti dalle funzioni della libreria di sistema `<stdio.h>`.

- Standard Output, `printf()`: la funzione utilizzata per inviare flussi di dati in uscita dal programma verso lo schermo del computer. Permette di visualizzare stringhe di testo e i valori memorizzati nelle variabili, formattandoli secondo le specifiche richieste dal programmatore.
- Standard Input, `scanf()`: la funzione deputata alla lettura dei dati immessi dall'utente, tipicamente tramite la tastiera. Arresta l'esecuzione del programma, interpreta i caratteri digitati e provvede a memorizzarli direttamente all'interno delle celle di memoria riservate alle variabili.

Abbiamo avuto modo di vedere il funzionamento dell'istruzione `printf()` in precedenza. Adesso, per incrementare ulteriormente il livello di competenze, è essenziale analizzare l'altra funzione di base: la `scanf()`. Sebbene il suo utilizzo presenti alcune complessità che verranno esaminate in seguito, essa rappresenta il primo strumento per implementare l'interazione dell'utente con il programma. Nello specifico, consente di acquisire un input esterno leggendo le informazioni inserite tramite la tastiera:

```
scanf("%d", &base);
```

Questa istruzione acquisisce un valore numerico dallo standard input (tastiera) e lo memorizza nella variabile base. Il segnaposto `%d` impone alla funzione di interpretare la sequenza di caratteri digitati dall'utente come un numero intero decimale. Con `&base` si ricava l'indirizzo di memoria della variabile base.

Vediamo un primo esercizio.

Esercizio 1

Scrivere un programma in C che permetta di calcolare l'area di un rettangolo. La base e l'altezza devono essere letti da tastiera.

```c
#include <stdio.h>
int main(void) {
  int base, altezza;
  printf("\nInserisci la base: ");
  scanf("%d", &base);
  printf("\nInserisci l'altezza: ");
  scanf("%d", &altezza);
  printf("\nL'area vale: %d",base * altezza);
  return 0;
}
```

Questo programma acquisisce in input da tastiera due numeri interi (`base` e `altezza`) e calcola l'area di un rettangolo. La sintassi della `scanf` presenta analogie con quella della `printf`, in quanto richiede di specificare il formato del dato atteso (`%d` in questo caso, trattandosi di un valore intero).

È tuttavia fondamentale notare la presenza del simbolo `&` anteposto al nome della variabile. Questo operatore, come già anticipato, definisce l'indirizzo di memoria della variabile. Il motivo di questa sintassi risiede nel fatto che nel linguaggio C il passaggio degli argomenti alle funzioni avviene esclusivamente per valore. Di conseguenza, se si desidera che una funzione (come la `scanf`) modifichi il contenuto di una variabile e non lavori su una sua copia, è necessario passarle direttamente l'indirizzo di memoria della variabile stessa, indicandole così in quale cella scrivere il dato acquisito. Questo concetto verrà approfondito nei capitoli dedicati alle funzioni e ai puntatori.

Importante

Cosa succede se non uso il simbolo `&`? Se non utilizzo il simbolo `&` il programma si comporta in modo *anomalo*. Il compilatore di norma non blocca la generazione dell'eseguibile, ma l'esecuzione del programma produce un comportamento anomalo che, nella quasi totalità dei casi, si traduce nel crash dell'applicazione. Perché? In sostanza si passa il valore memorizzato nella variabile e non il suo indirizzo ma `scanf` usa quel valore come se fosse un indirizzo e tenta di scrivere nella cella di memoria corrispondente. Il sistema operativo interviene e termina il programma per violazione di memoria (Segmentation Fault). Ricorda che un programma utente standard ha il permesso di leggere e scrivere esclusivamente all'interno dell'area di memoria (lo spazio di indirizzamento) che il sistema operativo gli ha riservato all'avvio.

Validazione dell'input

Un programma robusto dovrebbe validare l'input che riceve dall'esterno. Validare l'input significa verificarne il tipo, l'intervallo e la dimensione prima di utilizzarlo. In C questo controllo non avviene mai in automatico. Le conseguenze di una validazione assente o scorretta vanno dal comportamento imprevisto del programma fino a gravi vulnerabilità di sicurezza. Non è possibile trattare questi temi in modo completo in un corso di fondamenti, per ora è importante sapere che il problema esiste ed è reale.

5 Livello 3

Il presente capitolo approfondisce i meccanismi di deviazione del flusso di esecuzione nel linguaggio C, analizzando la sintassi e l'applicazione dei costrutti di selezione `if-else` e `switch`, le regole di valutazione delle espressioni logiche e l'impiego degli operatori relazionali e booleani.

5.1 Istruzioni di controllo decisionali

Nel linguaggio C, le istruzioni `if` e `switch` costituiscono i meccanismi fondamentali per le strutture condizionali. Il costrutto `if` opera valutando un'espressione logica generica e instradando il programma verso un determinato ramo in base al test effettuato. Lo `switch` implementa una struttura di selezione multivia confrontando il valore di una singola espressione (esclusivamente di tipo intero o carattere) con un elenco di costanti predefinite, semplificando la gestione di scenari a scelta multipla senza concatenare una sequenza di verifiche condizionali.

5.1.1 Istruzione if

L'istruzione di controllo più comune è il costrutto `if`, che rappresenta la forma più elementare di ramificazione. Tale costrutto valuta un'espressione logica, se l'espressione risulta vera (true), l'istruzione (o il blocco di istruzioni) ad essa associata viene eseguita, in caso contrario (false), l'istruzione viene omessa. È inoltre possibile combinare questo meccanismo con l'istruzione `else` per definire un ramo alternativo da eseguire qualora la condizione iniziale non sia soddisfatta.

È fondamentale tenere presente che nel linguaggio C la logica booleana viene valutata su base numerica (non esiste un tipo di dato booleano, almeno fino al C99). Il C99 introduce il tipo `_Bool` e l'header `<stdbool.h>`. Qualsiasi valore diverso da zero viene interpretato come vero (true), mentre il valore zero rappresenta il falso (false).

> **!** Importante
>
> In C, un blocco di istruzioni è rigidamente delimitato dall'uso delle parentesi graffe `{}`. Questo dettaglio sintattico è cruciale. Se si desidera eseguire un insieme di più istruzioni condizionate, l'uso delle graffe è obbligatorio. In assenza delle parentesi `{}`, il compilatore associerà all'istruzione `if` (o `else`) esclusivamente la prima istruzione immediatamente successiva, ignorando le righe seguenti, che verranno eseguite in modo sequenziale indipendentemente dall'esito del test.

5.1.2 Operatori relazionali e logici

Operatori relazionali e logici possono essere utilizzati per costruire espressioni più complesse. Gli operatori relazionali consentono di effettuare confronti tra variabili o espressioni.

- `==` Uguaglianza: verifica se i due operandi hanno lo stesso valore.
- `!=` Diversità: verifica se i due operandi hanno valori differenti.
- `<` Minore di: verifica se l'operando a sinistra è strettamente minore di quello a destra.
- `>` Maggiore di: verifica se l'operando a sinistra è strettamente maggiore di quello a destra.
- `<=` Minore o uguale a: verifica se l'operando a sinistra è minore o uguale a quello a destra.
- `>=` Maggiore o uguale a: verifica se l'operando a sinistra è maggiore o uguale a quello a destra.

La valutazione di un confronto genera un'espressione logica, ovvero un costrutto sintattico il cui esito definisce un valore di verità: VERO o FALSO. In C tale esito è concretamente il valore intero 1 se la condizione è verificata, 0 in caso contrario. L'espressione `5 > 3`, ad esempio, vale 1.

Per strutturare predicati condizionali complessi è necessario concatenare più espressioni. Questa operazione si realizza mediante gli operatori logici (o operatori booleani):

- `!` NOT (inversione logica): operatore unario che inverte il valore di verità dell'operando associato.
- `&&` AND (congiunzione logica): operatore binario che restituisce VERO esclusivamente se entrambi gli operandi sono veri.
- `||` OR (disgiunzione logica): operatore binario che restituisce VERO se almeno uno dei due operandi è vero.

Gli operatori `&&` e `||` godono di una proprietà importante, nota come valutazione a corto circuito. Il secondo operando viene valutato solo se strettamente necessario. Se il primo operando di un `&&` è falso, il risultato complessivo è certamente falso e il secondo operando viene del tutto ignorato. In modo speculare, se il primo operando di un `||` è vero, il risultato è certamente vero. Questo comportamento può essere sfruttato per scrivere condizioni sicure.

Vediamo due esercizi.

Esercizio 2

Scrivere un programma in C che permetta di valutare se un numero intero (letto da tastiera) è pari o dispari.

```c
#include <stdio.h>
int main(void){
  int n;
  printf("Inserisci un numero intero: ");
  scanf("%d",&n);
  if(n%2 == 0)
    printf("\n%d e' pari\n",n);
  else
    printf("\n%d e' dispari\n",n);
  return 0;
}
```

Esercizio 3

Scrivere un programma in C che permetta di verificare se un anno (letto da tastiera) è bisestile.

```c
#include <stdio.h>
int main(void){
   int anno;
   printf("Inserisci l'anno: ");
   scanf("%d",&anno);
   if ((anno%4 == 0 && anno%100 != 0) || anno%400 == 0)
      printf("\n%d e' bisestile\n", anno);
   else
      printf("\n%d non e' bisestile\n", anno);
   return 0;
}
```

5.1.3 Istruzione switch

Questo costrutto consente di selezionare uno specifico blocco di istruzioni in base al valore (`int` o `char`) assunto da una variabile o da un'espressione, verificata per uguaglianza rispetto a un elenco di valori costanti.

Vediamo lo stesso esercizio di prima.

Esercizio 4

Scrivere un programma in C che permetta di valutare se un numero intero (letto da tastiera) è pari o dispari.

```c
#include <stdio.h>
int main(void){
  int n;
  printf("Inserisci un numero intero \n");
  scanf("%d",&n);
  switch (n % 2) {
    case 0:
      printf("Il numero %d e' pari\n",n);
      break;
    default:
      printf("Il numero %d e' dispari\n",n);
      break;
  }
  return 0;
}
```

La sintassi del costrutto richiede due precisazioni importanti:

- `break`: ogni ramo `case` viene chiuso dall'istruzione `break`, che interrompe lo `switch` e trasferisce il controllo alla prima istruzione successiva al blocco. Il `break` non è obbligatorio, ma la sua omissione cambia radicalmente il comportamento del costrutto, una volta trovata la corrispondenza, l'esecuzione prosegue "a cascata" (fall-through) attraverso tutti i case successivi, ignorandone le etichette, fino al primo `break` incontrato o alla fine dello `switch`. Dimenticare un `break` è uno degli errori più classici e insidiosi del linguaggio C.
- `default`: è possibile aggiungere un ramo opzionale `default`, che viene eseguito quando il valore dell'espressione non corrisponde ad alcuna delle costanti elencate. Svolge un ruolo analogo a quello dell'`else` nel costrutto `if`.

6 Livello 4

Il presente capitolo analizza i costrutti iterativi del linguaggio C, esaminando la logica di esecuzione e l'equivalenza computazionale dei cicli `for`, `while` e `do-while`, l'impiego degli operatori di assegnamento composto e l'applicazione del cast di tipo nella gestione dei cicli.

6.1 Iterazione

Per sviluppare programmi di maggiore complessità, strutturando il codice sorgente in modo chiaro e conciso, è necessario introdurre le istruzioni di iterazione (o cicli). Quando l'algoritmo richiede l'esecuzione ripetuta di un'istruzione o di un blocco di istruzioni, si definisce un ciclo. La regola fondamentale, in questo contesto, impone di non duplicare mai la medesima istruzione nel codice.

6.1.1 Istruzione for

Il costrutto di iterazione utilizzato con maggiore frequenza nel linguaggio C è il ciclo `for`. Nella sua configurazione standard, un ciclo `for` presenta la seguente struttura sintattica:

```
for (i = 0; i < n; i++) {
    // Blocco di istruzioni da eseguire ripetutamente
}
```

Come si può notare, il costrutto racchiude tre campi distinti, separati dal carattere punto e virgola `;`:

- Primo Campo, *inizializzazione* (`i = 0`): la variabile `i` (preventivamente dichiarata nel programma) funge da variabile di controllo dell'iterazione. Questa istruzione viene eseguita una sola volta, esclusivamente all'avvio del ciclo, e determina lo stato iniziale del contatore. Una corretta inizializzazione è cruciale, infatti un errore in questa fase compromette l'integrità logica del programma senza che il compilatore segnali alcuna anomalia.
- Secondo Campo, *condizione di terminazione* (`i < n`): ospita un'espressione condizionale che viene valutata prima di intraprendere ciascuna iterazione del blocco di codice. Questo significa che il ciclo `for` (analogamente al ciclo `while`) è classificato come un costrutto a condizione preventiva, se l'espressione risulta falsa sin dal primo controllo, il blocco di codice interno non verrà eseguito mai.
- Terzo Campo, *aggiornamento* (`i++`): specifica l'incremento (o decremento) della variabile di controllo. Questa operazione viene eseguita dopo che tutte le istruzioni racchiuse nel blocco `{}` sono state portate a termine. L'espressione `i++` rappresenta la forma contratta della scrittura di assegnamento `i = i + 1`.

> **!** Importante
>
> Un fattore critico nella progettazione dei costrutti iterativi consiste nell'evitare la generazione di un **ciclo infinito**. Questa condizione anomala si verifica quando la logica delle istruzioni interne o del terzo campo non modifica la variabile di controllo in modo coerente, rendendo l'espressione condizionale del secondo campo costantemente vera.

Consideriamo il seguente esempio volto a illustrare l'applicazione pratica del ciclo `for`:

```c
#include <stdio.h>
int main(void) {
    int i, n, numero, somma = 0;
    float media;
    printf("Quanti numeri interi si desidera sommare? ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("\nNumero intero #%d: ", i);
        scanf("%d", &numero);
        somma += numero;
    }
    media = (float)somma / n;
    printf("\nLa media è pari a: %f\n", media);
    return 0;
}
```

In questo frammento di codice vengono introdotti alcuni elementi nuovi. L'accumulazione dei valori è espressa tramite la forma contratta (`+=`), derivata dalla combinazione di due operatori distinti. L'istruzione `somma += numero` è equivalente a `somma = somma + numero;`.

Sebbene la prima formulazione sia più compatta e ampiamente adottata nella pratica, richiede attenzione in quanto rende infatti meno esplicito il fatto che la variabile somma figuri anche sul lato destro dell'espressione. Per tale ragione, risulta fondamentale provvedere alla sua inizializzazione (impostandola a zero) prima del suo utilizzo, al fine di evitare l'elaborazione di valori imprevedibili residui della memoria RAM.

La seconda novità risiede nell'introduzione del costrutto di conversione di tipo o cast (`(float)somma`). Attraverso questo operatore, si modifica temporaneamente il tipo di dato della variabile somma all'interno dell'espressione di calcolo. Questo passaggio è di fondamentale importanza nel caso in esame perché in assenza di cast, la divisione tra `somma` e `n` verrebbe valutata dal compilatore come una divisione intera, troncando la parte decimale del risultato.

6.1.2 Istruzione while

Come accennato in precedenza, esiste almeno un altro costrutto di iterazione estremamente utile: il ciclo `while`. Nella maggior parte dei casi, questi due diversi cicli possono essere utilizzati in modo intercambiabile. La buona pratica prevede l'impiego del ciclo for quando il numero di iterazioni è noto a priori, riservando il `while` agli scenari in cui la terminazione dipende dal verificarsi di una determinata condizione dinamica.

Una delle differenze più evidenti risiede nella sintassi. Dopo la parola chiave `while`, all'interno delle parentesi `()`, deve essere specificata esclusivamente l'espressione condizionale. Nell'intestazione del costrutto non sono previsti spazi formali per l'inizializzazione della variabile di controllo o per il suo incremento/decremento. Questo non significa che tali fasi possano essere omesse, i medesimi passaggi logici devono semplicemente essere distribuiti in punti diversi del codice sorgente. Si consideri il seguente esempio, funzionalmente equivalente a quello analizzato in precedenza con il ciclo `for`:

```c
#include <stdio.h>
int main(void) {
    int i = 1, n, numero, somma = 0;
    float media;
    printf("Quanti numeri interi si desidera sommare? ");
    scanf("%d", &n);
    while (i <= n) {
        printf("\nNumero intero #%d: ", i);
        scanf("%d", &numero);
        somma += numero;
        i++;
    }
    media = (float)somma / n;
    printf("\nLa media è pari a: %f\n", media);
    return 0;
}
```

I due programmi sono equivalenti dal punto di vista computazionale ed è teoricamente possibile utilizzare indistintamente l'uno o l'altro costrutto.

6.1.3 Istruzione do-while

Sebbene sia di utilizzo meno frequente, il ciclo `do-while` costituisce una variante che non deve essere trascurata. La differenza sostanziale rispetto al ciclo `while` risiede nel momento in cui viene effettuata la verifica della condizione: nel `do-while` la condizione viene valutata dopo l'esecuzione del blocco di codice. Questo aspetto garantisce che le istruzioni racchiuse nel ciclo vengano eseguite almeno una volta, indipendentemente dal valore iniziale dell'espressione di controllo. Tale costrutto trova una collocazione ideale nelle procedure di validazione dell'input, dove è necessario acquisire un dato almeno una volta prima di poterne verificare la validità.

Per completezza, esistono due istruzioni in grado di alterare il normale flusso di un ciclo: `break` interrompe immediatamente l'iterazione e trasferisce il controllo alla prima istruzione successiva al ciclo, mentre `continue` salta direttamente all'iterazione successiva, omettendo le istruzioni rimanenti del blocco corrente. Si tratta di strumenti da utilizzare con cautela, poiché un impiego eccessivo rende più difficile seguire la logica del programma.

Esercizio 5

Scrivere un programma che permetta di inserire N voti, calcoli la media e conti il numero di voti sopra una soglia definita dall'utente.

```c
#include <stdio.h>
int main(void) {
  int n,i,voto,somma=0,soglia,sopra=0;
  printf("Quanti voti?");
  scanf("%d",&n);
  printf("Che soglia?");
  scanf("%d",&soglia);
  for(i=1;i<=n;i++) {
    printf("Inserisci voto %d: ",i);
    scanf("%d",&voto);
    somma+=voto;
    if(voto>soglia) ++sopra;
  }
  printf("\nLa media e': %.1f\n",(float)somma/n);
  printf("\n%d voti sopra %d", sopra,soglia);
  return 0;
}
```

`i++` oppure `++i`?

Si incontrano entrambe le forme `i++` (post-incremento) e `++i` (pre-incremento). Quando l'incremento costituisce un'istruzione a sé stante, come nel terzo campo di un ciclo `for`, le due forme sono perfettamente equivalenti. La differenza emerge solo quando l'espressione viene usata all'interno di un'espressione più ampia: `++i` incrementa la variabile e restituisce il valore già aggiornato, mentre `i++` restituisce il valore originale ed esegue l'incremento subito dopo.

7 Livello 5

Il presente capitolo introduce lo studio delle strutture dati lineari attraverso l'analisi degli array, esaminando i meccanismi di allocazione contigua in memoria RAM, le politiche di indicizzazione i vincoli architetturali legati alla manipolazione e alla copia dei vettori. In chiusura viene mostrato un esempio sull'uso di array bidimensionali (matrici).

7.1 Array monodimensionali

L'introduzione degli array monodimensionali (o vettori) estende significativamente le potenzialità di modellazione dei dati nel linguaggio C. Gli array costituiscono uno strumento estremamente potente. Qualora sia necessario gestire un insieme di variabili omogenee (tutte del medesimo tipo di dato), non è richiesto procedere alla loro dichiarazione individuale.

Analogamente a qualsiasi altra variabile, un array deve essere preventivamente dichiarato. La sintassi è la seguente:

```
int v[100];
```

Questa dichiarazione definisce un array monodimensionale denominato `v`, in cui ogni elemento condivide lo stesso identificatore e il medesimo tipo `int`. All'interno delle parentesi quadre `[]` è necessario indicare la dimensione. Nell'esempio proposto, verranno riservati nella memoria centrale 100 blocchi contigui della medesima dimensione (4 byte per ciascun intero).

> **!** Importante
>
> È fondamentale evidenziare che tali blocchi sono fisicamente adiacenti all'interno della memoria RAM. Di conseguenza, noto l'indirizzo del primo elemento, è possibile accedere in modo sequenziale a tutti gli altri elementi semplicemente applicando una traslazione del corretto numero di byte per raggiungere l'elemento successivo. Torneremo su questo aspetto dopo l'introduzione del concetto di **puntatore**.

L'accesso o la selezione di uno specifico elemento avviene mediante l'utilizzo di un indice, secondo la notazione `v[5]` (che in questo caso identifica l'elemento associato all'indice 5). Si ricordi che nel linguaggio C l'indicizzazione è basata sullo zero. Il primo elemento dell'array è associato all'indice zero (ovvero, `v[0]`). Di conseguenza, se l'array è strutturato per contenere 100 elementi, l'ultimo di essi sarà accessibile tramite l'indice 99 (`v[99]`). È necessario prestare la massima attenzione a non effettuare accessi al di fuori dell'intervallo valido, il quale spazia da 0 a N-1 (dove N rappresenta la dimensione dell'array). Il compilatore di norma non segnala alcun errore in fase di compilazione, ma tale scenario genera un comportamento pericoloso, in quanto il programma andrebbe a operare sul contenuto di blocchi di memoria adiacenti di cui non si ha il controllo. Infine, si raccomanda di evitare l'uso di costanti numeriche fisse per specificare la dimensione dell'array all'interno del codice. È preferibile ricorrere alla direttiva al preprocessore `#define N 100`. Tale

approccio permette di modificare centralmente la capacità dell'intera struttura dati qualora si renda necessario un successivo ridimensionamento.

> Buona pratica: evitare i "Numeri Magici"
>
> Scrivere direttamente valori numerici fissi nella dichiarazione degli array (es. `int v[100];`) è una pratica fortemente sconsigliata. Rende il codice rigido e costringe a modificare manualmente ogni singola occorrenza del numero qualora la dimensione debba variare in futuro.
> La soluzione standard in C prevede l'uso della direttiva al preprocessore.

Si consideri il seguente semplice esempio volto a illustrare l'utilizzo di un array:

```c
#include <stdio.h>
#define NVOTI 10
int main(void) {
    int i, n, voti[NVOTI], somma = 0;
    float media;
    printf("Quanti voti si desidera inserire? ");
    scanf("%d", &n);
    for (i = 0; i < n; ++i) {
        printf("\nInserisci il voto #%d: ", i + 1);
        scanf("%d", &voti[i]);
        somma += voti[i];
    }
    media = (float)somma / n;
    printf("\nLa media è pari a: %f\n", media);
    return 0;
}
```

La differenza principale rispetto all'Esercizio 5 del capitolo precedente risiede nel fatto che, al termine del ciclo `for`, tutti i voti rimangono memorizzati all'interno dell'array, ciascuno in un blocco di memoria specifico e consecutivo.

È sempre compito del programmatore dimensionare l'array in modo che possa ospitare un numero sufficiente di elementi. Qualora la capacità definita inizialmente risultasse insufficiente, sarà sufficiente modificare il valore all'interno della direttiva `#define`.

Si ricordi che **non è consentito** assegnare direttamente un intero array a un altro array tramite un'istruzione di assegnamento diretto del tipo `v = w;`. Il nome di un array, infatti, è sinonimo dell'indirizzo del suo primo elemento e tale indirizzo non è modificabile. Qualora si renda necessario copiare il contenuto di un array in un altro, è obbligatorio procedere alla copia puntuale di ciascun elemento, operando all'interno di un ciclo.

Array e puntatori: la regola e le eccezioni

Nella quasi totalità dei contesti il nome dell'array decade automaticamente all'indirizzo del suo primo elemento.
Ci sono soltanto pochi contesti in cui il decadimento non avviene. Esempio:

- `sizeof(v)` restituisce la dimensione dell'intero array (es. 400 byte), non quella di un puntatore (tipicamente 8 byte su 64 bit).
- `&v` produce un puntatore di tipo `int (*)[N]`, cioè un puntatore all'array nella sua interezza, non al primo elemento.

Per questo motivo, benché nella pratica il nome dell'array si comporti pressoché sempre come un puntatore costante al primo elemento, array e puntatori restano tipi distinti.

Per garantire un controllo sulle dimensioni dell'array, costituisce buona pratica verificare che il programma non accetti un numero di elementi superiore alla capacità massima allocata. Si analizzi il seguente esempio:

```c
#include <stdio.h>
#define N 10
int main(void) {
    int i, n, v[N], somma = 0;
    do {
        printf("Dimensione? ");
        printf("Deve essere compresa nell'intervallo 1-10! ");
        scanf("%d", &n);
    } while(n < 1 || n > N);
    for(i = 0; i < n; ++i) {
        printf("\nInserisci l'intero #%d: ", i + 1);
        scanf("%d", &v[i]);
        somma += v[i];
    }
    printf("\nLa somma è: %d\n", somma);
    return 0;
}
```

Importante

Il C non effettua alcun controllo sui limiti degli array durante l'esecuzione. In altre parole, è possibile leggere o scrivere un elemento al di fuori dell'intervallo valido senza ricevere alcun messaggio di errore. Il programma accederà alla memoria adiacente, con conseguenze imprevedibili.
È compito del programmatore garantire che tutti gli accessi avvengano entro i limiti e che gli array siano abbastanza grandi da contenere ciò per cui sono stati creati.

7.1.1 Altra inizializzazione di un array

Un array può essere inizializzato contestualmente alla dichiarazione, elencando i valori tra parentesi graffe:

```c
int v[5] = {3, 7, 12, 5, 9};
```

La sintassi prevede alcune varianti utili:

- Inizializzazione parziale: se i valori elencati sono meno della dimensione dichiarata, gli elementi rimanenti vengono automaticamente posti a zero. La scrittura `int v[100] = {0};` è quindi standard per ottenere un array interamente di zeri.
- Dimensione implicita: omettendo il numero tra parentesi quadre, è il compilatore a dedurre la dimensione contando gli inizializzatori. La dichiarazione `int v[] = {3, 7, 12};` definisce un array di 3 elementi.

7.1.2 Le stringhe

In C, le stringhe sono strutturate come array di elementi di tipo `char` la cui terminazione è marcata dal carattere nullo `\0`. Questo delimitatore consente di manipolare i flussi testuali scorrendoli in modo indipendente dalla dimensione allocata, ma impone di riservare sempre un byte aggiuntivo in fase di dichiarazione rispetto alla lunghezza effettiva del testo.

Sebbene sia possibile implementare manualmente le routine di copia e concatenazione mediante cicli iterativi che scansionano i vettori fino al raggiungimento del terminatore, la gestione ottimale e sicura della manipolazione testuale si affida alle funzioni native della libreria `<string.h>` (quali `strcpy`, `strcat`, `strlen` e `strcmp`).

7.2 Array bidimensionali

Le matrici nel linguaggio C estendono il concetto di array monodimensionale introducendo strutture dati bidimensionali, dichiarate tramite la sintassi `M[NR][NC]` e indicizzate specificando riga e colonna mediante la notazione `M[i][j]`.

La manipolazione di tali strutture si realizza attraverso l'annidamento di cicli iterativi. Vediamo un esercizio che mostri il funzionamento di un array bidimensionale.

Esercizio 6

Scrivere un programma che calcoli la somma delle righe e delle colonne di una matrice quadrata di dimensione NxN

```c
#include <stdio.h>
#define DIM 100
int main(void) {
    int i,j,m1[DIM][DIM],rsum[DIM],csum[DIM],n;
    do  {
      printf("Dimensione matrice quadrata: \n");
      scanf("%d", &n);
    } while(n<1 || n>DIM);
    printf("Inserisci elementi della matrice:\n");
    for(i=0;i<n;i++) {
        for(j=0;j<n;j++) {
            printf("elemento - [%d],[%d] : ",i,j);
            scanf("%d",&m1[i][j]);
        }
    }
    for(i=0;i<n;i++) {
        rsum[i]=0;
        for(j=0;j<n;j++)
            rsum[i]=rsum[i]+m1[i][j];
    }
    for(i=0;i<n;i++) {
        csum[i]=0;
        for(j=0;j<n;j++)
            csum[i]=csum[i]+m1[j][i];
    }
    printf("La somma righe e colonne vale:\n");
    for(i=0;i<n;i++) {
        for(j=0;j<n;j++)
            printf("%4d",m1[i][j]);
        printf("%8d",rsum[i]);
        printf("\n");
    }
    printf("\n");
    for(j=0;j<n;j++) {
        printf("%4d",csum[j]);
    }
    printf("\n\n");
    return 0;
}
```

E in Python?

A livello sintattico, l'accesso a un elemento tramite indice appare identico in C e in Python (es. `v[0]`). Tuttavia, dal punto di vista della gestione della memoria RAM, i due linguaggi operano secondo paradigmi molto differenti. Il confronto che segue evidenzia il trade-off tra il controllo diretto dell'hardware (in C) e l'astrazione software (in Python).

- **L'approccio in C (allocazione contigua):** che si tratti di un array **statico** (allocato sullo Stack) o di un array **dinamico** (allocato sull'Heap tramite `malloc()`, lo vedremo meglio in seguito), il linguaggio C garantisce sempre la **contiguità fisica** dei blocchi di memoria. Gli elementi adiacenti dell'array occupano celle adiacenti della RAM.
- **L'approccio in Python (struttura referenziale):** una lista in Python non contiene direttamente i dati (i numeri o le stringhe), ma contiene una sequenza di puntatori contigui che indirizzano gli oggetti effettivi. I dati reali possono essere dislocati in zone della RAM completamente sparse e distanti tra loro.

Questa architettura del C impone che l'array sia rigidamente **omogeneo** (tutti gli elementi devono condividere lo stesso tipo di dato, es. `int`). Questo vincolo permette alla CPU di calcolare l'indirizzo di memoria di un qualsiasi elemento eseguendo una semplice operazione di aritmetica dei puntatori a livello hardware:

$$\text{Indirizzo}(v[i]) = \text{Indirizzo Base} + (i \times \text{dimensione\_tipo})$$

In Python, la lista è **eterogenea** perché potendo memorizzare puntatori a oggetti qualsiasi, una singola lista può contenere contemporaneamente interi, stringhe e float.

8 Livello 6

8.1 Le strutture

Come appena visto, gli array sono collezioni di elementi omogenei (caratterizzati dal medesimo tipo di dato). Attraverso le **strutture** (`struct`), possiamo invece definire tipi di dato nuovi e personalizzati. Le variabili dichiarate di tipo struttura possono aggregare diverse componenti (campi), ciascuna contraddistinta da uno specifico tipo di dato. Di seguito un esempio elementare di **definizione** di una struttura e della successiva **dichiarazione** di una variabile:

```
struct studente {
    int matricola;
    char nome[DIM_NOME];
    char cognome[DIM_NOME];
};
struct studente stud;
```

L'accesso ai singoli campi della struttura avviene mediante l'utilizzo dell'operatore punto (`.`), secondo la notazione: `stud.matricola`, `stud.nome` e `stud.cognome`. Per popolare una variabile di tipo struttura con dati acquisiti dallo standard input, si consideri il seguente frammento di codice:

```
#include <stdio.h>
#define DIM_NOME 20

struct studente {
    int matricola;
    char nome[DIM_NOME];
    char cognome[DIM_NOME];
};

int main(void) {
    struct studente stud;
    printf("\nNome: ");
    scanf("%s", stud.nome);
    printf("\nCognome: ");
    scanf("%s", stud.cognome);
    printf("\nMatricola: ");
    scanf("%d", &stud.matricola);
    printf("\n\nDati studente: ");
    printf("%s %s, Matricola: %d\n", stud.nome,
      stud.cognome, stud.matricola);
    return 0;
}
```

Si noti che quando si utilizza la funzione `scanf` per l'acquisizione di una stringa (array di caratteri), non è necessario anteporre l'operatore `&` al nome del campo. Questo avviene perché la `scanf` richiede come argomento un indirizzo di memoria e l'identificatore di un array (e dunque di una stringa) rappresenta nativamente l'indirizzo al suo primo elemento. Di conseguenza, l'omissione dell'operatore `&` è corretta.

8.1.1 Gestione sicura delle stringhe: `scanf` e `fgets`

L'uso della `scanf` per la lettura di stringhe da tastiera è fortemente sconsigliato per due ragioni fondamentali:

1. Non consente di verificare esplicitamente i limiti fisici dell'array di destinazione, esponendo il sistema a gravi vulnerabilità di **buffer overflow** qualora l'utente inserisca una stringa più lunga della capacità allocata.
2. L'acquisizione del flusso viene interrotta non appena viene rilevato un carattere di spaziatura (spazio vuoto, tabulazione o ritorno a capo), rendendo impossibile l'acquisizione di stringhe composte da più parole.

Una soluzione consiste nell'impiego della funzione di libreria `fgets`. Si analizzi la versione aggiornata del codice:

```c
#include <stdio.h>
#define DIM_NOME 20

struct studente {
    int matricola;
    char nome[DIM_NOME];
    char cognome[DIM_NOME];
};

int main(void) {
    struct studente stud;
    printf("\nNome: ");
    fgets(stud.nome, DIM_NOME, stdin);
    printf("\nCognome: ");
    fgets(stud.cognome, DIM_NOME, stdin);
    printf("\nMatricola: ");
    scanf("%d", &stud.matricola);
    printf("\n\nDati studente: ");
    printf("%s %s, Matricola: %d\n", stud.nome,
      stud.cognome, stud.matricola);
    return 0;
}
```

Come si può osservare, la funzione `fgets` richiede come secondo argomento la dimensione massima del buffer (`DIM_NOME`), garantendo che a runtime (durante l'esecuzione del programma) non scriva mai oltre il limite allocato. Il terzo parametro, `stdin` (*standard input*), specifica che il canale di comunicazione attivo è la tastiera. Nei capitoli successivi verrà mostrato come la medesima funzione possa essere applicata per la lettura di flussi di dati da file.

8.2 Strutture dati complesse: array di strutture

Oltre al caso appena esaminato in cui una **struttura contiene degli array**, il linguaggio C permette di elevare il livello di astrazione definendo **array di strutture** o **strutture annidate** (strutture che ospitano altre strutture al loro interno). Il seguente programma illustra la gestione di un elenco di record di studenti tramite un array di `struct`:

```c
#include <stdio.h>
#define DIM_NOME 20
#define DIM_STUDI 100

struct studente {
    int matricola;
    char nome[DIM_NOME];
    char cognome[DIM_NOME];
};

int main(void) {
    struct studente elenco[DIM_STUDI];
    int n_studenti;
    char invio;
    printf("\nQuanti studenti si desidera inserire? ");
    scanf("%d", &n_studenti);
    printf("\nInserimento dati:\n");
    for(int i = 0; i < n_studenti; ++i) {
        printf("\nStudente - %d: ", i + 1);
        printf("\nNome: ");
        scanf("%c", &invio); // Consuma il carattere newline residuo
        fgets(elenco[i].nome, DIM_NOME, stdin);
        printf("Cognome: ");
        fgets(elenco[i].cognome, DIM_NOME, stdin);
        printf("Matricola: ");
        scanf("%d", &elenco[i].matricola);
    }
    printf("\n\nRegistro Studenti:\n");
    for(int i = 0; i < n_studenti; ++i) {
        printf("%s %s - Matricola: %d\n", elenco[i].nome,
          elenco[i].cognome, elenco[i].matricola);
    }
    return 0;
}
```

> ⚠ La Gestione del buffer di input
>
> Si presti particolare attenzione all'inserimento dell'istruzione `scanf("%c", &invio);` antecedente alla `fgets`. Quando si acquisisce un valore numerico tramite `scanf`, il carattere di ritorno a capo (`\n`) generato dalla pressione del tasto Invio rimane memorizzato nel buffer dello standard input. Poiché la funzione `fgets` interrompe la lettura non appena incontra un newline, se il buffer non venisse preventivamente ripulito (o "consumato"), la `fgets` successiva leggerebbe immediatamente il carattere `\n` residuo, interpretandolo come una stringa vuota e saltando l'acquisizione del dato.

8.3 `struct` o `typedef`

Nel linguaggio C, la definizione e la successiva dichiarazione di una struttura possono essere affrontate con due paradigmi differenti: l'uso esplicito della parola chiave `struct` o l'impiego del costrutto `typedef`.

8.3.1 1. L'approccio standard (esplicito)

Nel primo caso, si definisce il modello della struttura e si dichiara la variabile ripetendo la parola chiave `struct` ad ogni istanza:

```
// Definizione
struct studente {
    int matricola;
    char nome[NOME];
    char cognome[NOME];
};

// Dichiarazione
struct studente mio_studente;
```

8.3.2 2. L'approccio con `typedef` (alias)

Nel secondo caso, si utilizza la parola chiave `typedef` per creare un vero e proprio *alias* (un nuovo identificatore di tipo) per la struttura. Questo approccio permette di omettere la parola `struct` durante la dichiarazione delle variabili, rendendo la sintassi più compatta:

```
// Definizione
typedef struct {
    int matricola;
    char nome[NOME];
    char cognome[NOME];
} studente;

// Dichiarazione
studente mio_studente;
```

8.3.3 Perché preferire `struct`?

I principali vantaggi dell'utilizzo diretto di `struct` includono:

- **Leggibilità e trasparenza:** mantenere visibile la parola chiave `struct` impedisce di "mascherare" la natura del dato. Chi analizza il codice comprende immediatamente che sta manipolando una struttura.
- **Chiarezza del dominio utente:** il rischio intrinseco del `typedef` è quello di creare un'astrazione opaca. Per comprendere questo rischio, si consideri la dichiarazione `typedef int numero;`. Essa permette di scrivere `numero x = 10;`, nascondendo tuttavia al lettore il fatto che `x` sia banalmente un intero.
- **Compatibilità storica:** l'uso esplicito garantisce la massima aderenza e retrocompatibilità con i vecchi standard del linguaggio, facilitando l'integrazione e la manutenzione.

9 Livello 7

9.1 I puntatori

In C, i puntatori sono onnipresenti e vengono impiegati in quasi ogni ambito della programmazione.

> **Definizione:** Un puntatore è una variabile destinata a memorizzare l'indirizzo di memoria (la coordinata fisica nella RAM) di un'altra variabile.

Come verrà analizzato dettagliatamente, i puntatori sono indispensabili per manipolare gli array, modificare il contenuto delle variabili passate alle funzioni, gestire i file, governare l'allocazione dinamica della memoria e strutturare collezioni di dati complesse come le liste concatenate.

La sintassi per la dichiarazione di un puntatore richiede l'uso dell'operatore `*` anteposto al nome della variabile:

```
int *mio_puntatore;
```

La dichiarazione definisce l'identificatore del puntatore e, mediante il tipo specificato (in questo caso `int`), stabilisce il tipo di dato della variabile puntata. È possibile dichiarare puntatori a qualunque tipo di dato.

Si analizzi un primo esempio essenziale per comprendere il funzionamento di base dei puntatori:

```
#include <stdio.h>
int main(void) {
    int n = 10;
    int *p;
    p = &n; // Inizializzazione: p memorizza l'indirizzo di n
    printf("%d %d\n", n, *p);
    return 0;
}
```

In questo frammento, alla dichiarazione `int *p;` segue l'inizializzazione `p = &n;`. Quest'ultima istruzione mappa il puntatore `p` sulla variabile `n`, trasferendo l'indirizzo di quest'ultima all'interno del puntatore. Da questo momento, `p` contiene l'indirizzo in RAM di `n`, e diventa possibile modificare il valore di `n` per via indiretta tramite l'espressione `*p` (operatore di *dereferenziazione*).

9.1.1 Accesso alle strutture tramite puntatori

I puntatori possono indirizzare variabili composte come le `struct`. Per accedere ai campi di una struttura attraverso un puntatore dedicato, il C introduce l'operatore freccia (`->`), che sostituisce la combinazione dell'operatore di dereferenziazione e dell'operatore punto (`(*p).campo`).

```c
#include <stdio.h>
#define DIM_NOME 20
struct studente {
    int matricola;
    char nome[DIM_NOME];
    char cognome[DIM_NOME];
};
int main(void) {
    struct studente stud;
    struct studente *mio_p;
    mio_p = &stud; // Il puntatore referenzia la struttura
    printf("\nNome: ");
    fgets(mio_p->nome, DIM_NOME, stdin);
    printf("\nCognome: ");
    fgets(mio_p->cognome, DIM_NOME, stdin);
    printf("\nMatricola: ");
    scanf("%d", &mio_p->matricola);
    printf("\n\nDati studente: ");
    printf("%s %s, Matricola: %d\n", stud.nome,
      stud.cognome, stud.matricola);
    return 0;
}
```

L'istruzione `mio_p = &stud;` vincola formalmente il puntatore all'istanza della struttura. Di conseguenza, le funzioni di input agiscono direttamente sui membri della `struct` originaria operando per via indiretta tramite il puntatore.

9.1.2 Il ruolo dei puntatori nelle liste concatenate

Come vedremo più avanti, i puntatori costituiscono il componente fondamentale per l'implementazione delle **liste concatenate** (*linked lists*). Anche nella loro forma più elementare, ovvero in regime di allocazione statica della memoria, le liste sono strutturate attraverso i puntatori. Sussistono tre ragioni fondamentali che rendono l'uso dei puntatori vincolante nelle liste:

1. A differenza degli array, gli elementi (nodi) di una lista concatenata non sono necessariamente allocati in blocchi di memoria fisicamente adiacenti. L'unico meccanismo per determinare la posizione del nodo successivo resta quello di memorizzarne l'indirizzo all'interno del nodo precedente. Ogni nodo deve pertanto integrare un campo puntatore.
2. La lista è identificata dal puntatore che indirizza il suo primo elemento (la "testa" della lista). Lo smarrimento di questo puntatore comporta l'impossibilità di scorrere la lista. Nell'esempio che segue i nodi sono variabili locali dotate di un proprio nome e restano comunque raggiungibili, quando però i nodi verranno allocati dinamicamente (come

accade nelle liste reali), la perdita della testa renderà la loro memoria irraggiungibile e non più rilasciabile (*memory leak*, ci torneremo nel capitolo sugli oggetti dinamici).
3. La scansione della lista richiede l'individuazione del nodo terminale. È obbligatorio imporre che il campo puntatore dell'ultimo nodo sia impostato sulla costante **`NULL`**, un valore convenzionale che attesta la fine della lista.

Si consideri un esempio di lista concatenata statica, in cui tutti i nodi sono preventivamente allocati sullo stack:

```c
#include <stdio.h>
struct lista {
    int num;
    struct lista *next; // Puntatore al nodo successivo
};
int main(void) {
    struct lista val1, val2, val3;
    struct lista *punt_list;
    // Configurazione dei dati e concatenazione dei nodi
    val1.num = 10;
    val1.next = &val2;
    val2.num = 20;
    val2.next = &val3;
    val3.num = 30;
    val3.next = NULL; // Nodo terminale
    punt_list = &val1; // Il puntatore individua l'inizio della lista
    // Ciclo di scansione sequenziale
    while(punt_list != NULL) {
        printf("%d ", punt_list->num);
        punt_list = punt_list->next; // Traslazione al nodo successivo
    }
    printf("\n");
    return 0;
}
```

Le liste concatenate offrono un'elevata flessibilità. Le operazioni di inserimento o rimozione di un nodo intermedio non richiedono lo spostamento fisico dei dati in memoria, ma si risolvono mediante il semplice aggiornamento dei puntatori dei nodi coinvolti. Negli array questa medesima operazione presenta una complessità superiore, poiché richiede lo shift (scorrimento) di tutti gli elementi successivi alla posizione d'esame per preservare la contiguità degli elementi.

> **i** Una precisazione terminologica
>
> In questo esempio i nodi sono dichiarati come variabili locali e risiedono quindi sullo stack. Nel gergo comune si parla spesso di "allocazione statica" per indicare tutto ciò che non viene allocato dinamicamente, ma è bene sapere che lo standard C distingue tre tipologie di memorizzazione: automatica (le variabili locali, sullo stack), statica in senso proprio (le variabili globali e quelle dichiarate con la parola chiave `static`) e dinamica (la memoria richiesta a runtime con `malloc`, che vedremo nel capitolo sugli oggetti dinamici).

9.1.3 Relazione tra array e puntatori

Esiste una stretta relazione tra array e puntatori. È possibile associare un puntatore esterno ad un array e scorrere gli elementi dell'array applicando le regole dell'**aritmetica dei puntatori**.

Data la seguente dichiarazione:

```
int v[N];
int *v_ptr;

v_ptr = &v[0]; // Il puntatore indirizza l'elemento iniziale
```

Il puntatore `v_ptr` memorizza l'indirizzo del primo elemento dell'array `v`, consentendo di interagire con l'intera struttura lineare.

```
#include <stdio.h>
int main(void) {
    int i, v[3] = {1, 2, 3};
    int *v_ptr;
    for(i = 0; i < 3; i++) printf("%d ", v[i]); // Output: 1 2 3
    printf("\n");
    v_ptr = v; // Equivalente a: v_ptr = &v[0];
    *(v_ptr + 1) = 10; // Modifica del secondo elemento
    for(i = 0; i < 3; i++) printf("%d ", v[i]); // Output: 1 10 3
    printf("\n");
    return 0;
}
```

In Figura 9.1, vediamo l'organizzazione dell'array `v` e del puntatore `v_ptr` (successivamente all'istruzione `v_ptr = v;`) in memoria RAM.

L'espressione `*(v_ptr + 1) = 10;` dimostra come modificare il contenuto del secondo elemento dell'array. La notazione `v_ptr + 1` non incrementa l'indirizzo di un singolo byte, lo spostamento (offset) viene automaticamente scalato sulla dimensione del tipo puntato. Se l'array memorizza interi, lo scostamento fisico sarà tipicamente di 4 byte, se l'array fosse composto da caratteri (`char`), lo scostamento sarebbe di 1 byte. In generale, l'espressione `v_ptr + i` corrisponde all'indirizzo base aumentato di $i \times$ `sizeof(tipo)` byte. L'operatore `*` garantisce poi l'accesso diretto al valore memorizzato a quell'indirizzo.

In virtù di queste proprietà, all'interno del codice le notazioni vettoriali e le espressioni basate sui puntatori risultano del tutto equivalenti dal punto di vista computazionale:

$$v[i] \iff *(v\_ptr + i)$$

Infine, è possibile impiegare gli operatori di incremento e decremento (`v_ptr++`, `v_ptr--`) per spostare dinamicamente il puntatore tra gli elementi dell'array. Si raccomanda di considerare che tali operazioni modificano il valore memorizzato nella variabile puntatore stessa e quindi al termine delle elaborazioni, `v_ptr` non farà più riferimento all'elemento iniziale dell'array, a meno di non procedere a un esplicito riallineamento (`v_ptr = v;`).

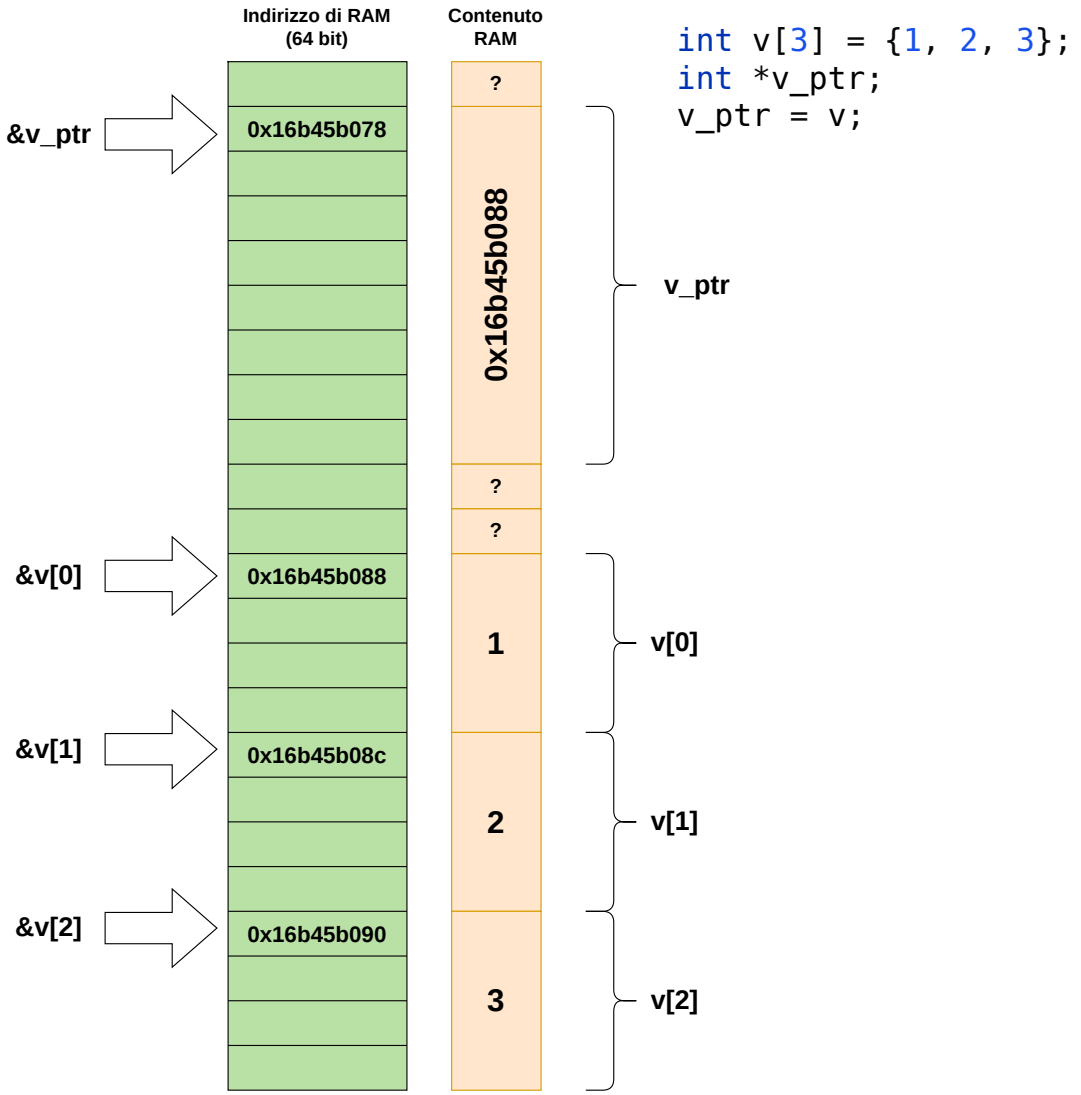

Figura 9.1: Schema dell'organizzazione in RAM di un array. L'array `v` occupa 12 byte, in quanto è costituito da tre interi da 4 byte l'uno. Gli elementi dell'array sono adiacenti in RAM. Si noti che la differenza tra gli indirizzi di ogni elemento è pari a 4 byte (nella figura gli indirizzi sono espressi in esadecimale). A seguito dell'istruzione `v_ptr = v;`, il puntatore `v_ptr` contiene al suo interno l'indirizzo del primo elemento dell'array `v`. Tramite l'aritmetica dei puntatori è possibile accedere all'array `v` tramite il puntatore `v_ptr`.

! Il decadimento

Come già visto nel capitolo sugli array, in quasi tutti i contesti il nome di un array decade automaticamente nell'indirizzo del suo primo elemento ($v = \&v[0]$), per questo `v_ptr = v` è del tutto equivalente a `v_ptr = &v[0]`, e quando si passa un array a una funzione la funzione riceve un indirizzo, non una copia dell'array. Il nome dell'array non è però riassegnabile né incrementabile, operazioni come `v = v + 1` o `v++` non sono ammesse dal compilatore.
Il decadimento non si verifica soltanto in pochissimi contesti eccezionali: `sizeof(v)`, che restituisce la dimensione dell'intero array e non quella di un puntatore e `&v` produce un puntatore di tipo `int (*)[N]`. Al di fuori di queste eccezioni, il nome dell'array e un puntatore al suo primo elemento si comportano in modo equivalente nelle espressioni.

10 Livello 8

10.1 Le funzioni

Le funzioni sono sottoprogrammi, ovvero **blocchi di codice riutilizzabili** concepiti per eseguire **una** specifica funzionalità. Rappresentano il primo passo fondamentale verso la **modularità** del software. Nel linguaggio C è possibile sia utilizzare le funzioni standard messe a disposizione dalle librerie di sistema, sia implementare funzioni personalizzate.

Una funzione dovrebbe comportarsi come una *black box*, deve esporre precise funzionalità all'esterno, nascondendo al contempo i dettagli algoritmici della propria implementazione. L'interazione con una funzione prevede, nella maggior parte dei casi, il passaggio di parametri in ingresso e la restituzione di un valore come risultato al termine dell'elaborazione.

Per integrare correttamente una funzione all'interno di un programma, è necessario articolare il codice in tre fasi distinte: la **dichiarazione** (o prototipo), la **chiamata** (che può essere effettuata nel `main` o in un altro sottoprogramma) e la **definizione**. Si analizzi questo esempio elementare:

```c
#include <stdio.h>
int somma(int, int); // Dichiarazione della funzione (prototipo)
int main(void) {
    int a = 1, b = 1, s;
    s = somma(a, b); // Chiamata della funzione
    printf("\nLa somma è: %d\n", s);
    return 0;
}

int somma(int c, int d) { // Definizione della funzione
    int s;
    s = c + d;
    return s;
}
```

La dichiarazione `int somma(int, int);` comunica al compilatore l'interfaccia della funzione prima del suo utilizzo effettivo, ne stabilisce il nome (`somma`), specifica il tipo del valore restituito (`int`) e definisce il numero e il tipo dei parametri accettati `(int, int)`. Per motivi di chiarezza descrittiva è consentito inserire dei nomi per i parametri anche nel prototipo, ma questi verranno ignorati dal compilatore.

La chiamata `s = somma(a, b);` eseguita nel `main` devia il flusso di esecuzione verso il blocco di codice della funzione. In questo contesto, le variabili `a` e `b` prendono il nome di **parametri attuali**.

La definizione, che inizia con l'intestazione `int somma(int c, int d)` ed è seguita dal blocco di istruzioni racchiuso tra parentesi graffe, contiene l'insieme delle operazioni da

svolgere. Le variabili `c` e `d` sono denominate **parametri formali** e si configurano come variabili *locali*, ovvero non sono visibili né accessibili all'esterno della funzione stessa.

Il linguaggio C adotta esclusivamente il meccanismo di **passaggio dei parametri per valore**, di conseguenza, la funzione **opera sempre su una copia** dei parametri attuali. All'atto della chiamata, i parametri formali vengono inizializzati con i valori dei corrispondenti parametri attuali, impedendo alla funzione di alterare il contenuto delle variabili originali. *Chiamata* e *definizione* devono concordare per **numero**, **tipo** e **ordine** dei parametri.

10.1.1 Passaggio tramite puntatori

Qualora si richieda che una funzione modifichi il valore di una variabile definita nel modulo chiamante, è necessario implementare un passaggio utilizzando i puntatori. Si consideri l'esempio classico dello scambio (*swap*) tra due variabili:

```c
#include <stdio.h>
void scambia(int *, int *); // Il prototipo accetta due indirizzi
int main(void) {
    int x = 0, y = 1;
    printf("\nPrima della chiamata: x = %d e y = %d", x, y);
    scambia(&x, &y); // Chiamata: si passano gli indirizzi delle variabili
    printf("\nDopo la chiamata: x = %d e y = %d\n", x, y);
    return 0;
}

void scambia(int *x, int *y) {
    int tmp;
    tmp = *x; // Salva il valore contenuto all'indirizzo puntato da x
    *x = *y;  // Assegna all'indirizzo puntato da x il valore presente in y
    *y = tmp; // Completa lo scambio
}
```

Come si può osservare, il ricorso ai puntatori consente alla funzione `scambia` di accedere indirettamente alle celle di memoria del `main` e di modificarne i valori originari. Si noti che tale meccanismo prescinde dagli identificatori utilizzati, non sussiste alcuna relazione tra le variabili `x` e `y` del `main` e i puntatori locali `x` e `y` definiti nell'argomento della funzione. Si provi per esempio a stampare gli indirizzi di memoria delle variabili `x` e `y` nel `main` e delle variabili `x` e `y` in `scambia`.

> ⚠ Variabili globali
>
> Si raccomanda caldamente di evitare l'impiego di variabili globali come canale implicito per il trasferimento di dati tra funzioni. Tale approccio compromette la modularità del software, ostacola le procedure di debugging e aumenta esponenzialmente il rischio di comportamenti anomali.

Un concetto cardine nella gestione delle funzioni è l'**ambito di visibilità** (*scope*) delle variabili, definito come la porzione di codice in cui una determinata variabile è dichiarata, riconosciuta e utilizzata.

10.1.2 Interazione tra funzioni e array

È frequente la necessità di sviluppare funzioni destinate a operare su strutture dati lineari (array). Se il passaggio di un singolo elemento dell'array (`v[i]`) avviene secondo le regole standard delle variabili, il trasferimento dell'intera struttura dati richiede un approccio differente. Nel linguaggio C non è consentito passare un array in blocco per valore, poiché ciò richiederebbe la duplicazione di ogni singolo elemento in memoria, determinando un overhead computazionale inaccettabile.

Per superare questo vincolo si passa alla funzione il **nome dell'array** (`v`). Tale operazione equivale a trasmettere l'indirizzo di memoria del suo primo elemento ($v = \&v[0]$). Una volta che la funzione ha acquisito l'indirizzo iniziale dell'array, può accedere a tutti gli elementi successivi applicando l'aritmetica dei puntatori. Poiché la funzione riceve esclusivamente un indirizzo, non ha modo di determinarne la dimensione, è pertanto indispensabile passare la dimensione dell'array come parametro aggiuntivo.

Il seguente esempio illustra l'implementazione di una funzione preposta alla ricerca del valore minimo all'interno di un vettore di interi:

```c
#include <stdio.h>
#define DIM 10
int calcola_minimo(int *, int);
int main(void) {
    int n, v[DIM], i, min;
    do {
        printf("Dimensione dell'array: \n");
        scanf("%d", &n);
    } while(n < 1 || n > DIM);
    printf("Inserisci i %d elementi.\n", n);
    for(i = 0; i < n; i++) {
        printf("Elemento con indice %d: ", i);
        scanf("%d", &v[i]);
    }
    min = calcola_minimo(v, n); // Chiamata: v trasmette l'indirizzo base
    printf("\nIl valore minimo è: %d\n", min);
    return 0;
}

int calcola_minimo(int *v, int n) {
    int i, min = *v; // Inizializzazione con il primo elemento dell'array
    for(i = 0; i < n; i++) {
        if(*(v + i) < min)
            min = *(v + i); // Accesso via aritmetica dei puntatori
    }
    return min;
}
```

Come mostrato, la funzione `calcola_minimo` accetta come parametri attuali l'indirizzo dell'array `v` e la sua dimensione `n`. Il parametro formale `int *v` è un puntatore locale che riceve l'indirizzo dell'array definito nel `main`. I due identificatori `v` appartengono a spazi di visibilità separati.

Si analizzi questo ulteriore esempio focalizzato sul calcolo della media, volto a chiarire la natura modificabile del parametro formale:

```c
#include <stdio.h>
#define DIM 10
float calcola_media(int *, int);
int main(void) {
    int n, v[DIM], i;
    float media;
    do {
        printf("Dimensione dell'array: \n");
        scanf("%d", &n);
    } while(n < 1 || n > DIM);
    printf("Inserisci i %d elementi.\n", n);
    for(i = 0; i < n; i++) {
        printf("Elemento con indice %d: ", i);
        scanf("%d", &v[i]);
    }
    media = calcola_media(v, n);
    printf("\nLa media è pari a: %.1f\n", media);
    return 0;
}

float calcola_media(int *v, int n) {
    int i;
    float somma = 0;
    for(i = 0; i < n; i++, v++) {
        somma += *v; // Dereferenziazione e incremento del puntatore locale
    }
    return somma / n;
}
```

In questo scenario, l'espressione di incremento `v++` inserita nel ciclo della funzione `calcola_media` viene eseguita correttamente e senza generare anomalie. Questo è possibile poiché il parametro formale `v` all'interno della funzione è una variabile puntatore locale (una copia dell'indirizzo originario) e non risente del vincolo di costanza che caratterizza invece l'identificatore dell'array nel `main`, il quale rimane immutato al termine delle elaborazioni.

Esercizio 7

Scrivere un programma in C che consenta, tramite l'uso di specifiche funzioni di caricare n elementi in un array e di stampare a video i valori inseriti.

```c
#include <stdio.h>
#define N 10
void carica_array(int, int *);
void stampa_array(int, int *);
int main(void) {
    int v[N],n;
    do{
        printf("\nDimensione array: ");
        scanf("%d",&n);
    }while(n<1 || n>N);
    printf("\nIndirizzo di v[0] nel main: %p ",&v[0]);
    carica_array(n,&v[0]); // Equivalente a carica_array(n,v);
    stampa_array(n,v);
    return 0;
}

void carica_array(int n, int *x){
    int i;
    printf("\nx e' inizializzata con l'indirizzo di v[0]: %p ", x);
    printf("\nLa variabile x e' locale e si trova in %p",&x);
    for(i=0;i<n;i++){
        printf("\nInserisci elemento: ");
        scanf("%d",x+i);
        // tramite x carico i valori nell'array v dichiarato nel main
    }
}
void stampa_array(int n, int *v){
    int i;
    printf("\nArray: ");
    for(i=0;i<n;i++){
        printf("%d ",*(v+i));
    }
}
```

10.2 L'importanza del tipo nei puntatori

In questo contesto e alla luce di quanto visto finora, ci si può chiedere: se un puntatore serve esclusivamente a memorizzare un indirizzo, perché il C impone di dichiarare un tipo specifico anziché utilizzare un tipo generico per tutti i puntatori? La risposta risiede nel fatto che il puntatore non si limita a indicare il punto di partenza nella RAM, ma fornisce al compilatore le **coordinate dimensionali** per due operazioni fondamentali:

1. **La dereferenziazione:** l'indirizzo salvato nel puntatore indica solo da *quale* byte iniziare a leggere. È il tipo del puntatore a dire al compilatore *quanti* byte consecutivi

devono essere letti. Un puntatore `int *` istruirà la CPU a leggere 4 byte contigui, mentre un `char *` dirà di leggerne soltanto 1.

2. **L'aritmetica dei puntatori:** quando si esegue un'operazione di incremento come `p + 1`, il compilatore calcola l'offset in base alla dimensione del tipo puntato. Un `int *` traslerà l'indirizzo in avanti di 4 byte, mentre un `char *` lo sposterà di un singolo byte.

Per toccare con mano questo meccanismo, analizziamo il seguente frammento di codice in cui lo stesso indirizzo di memoria viene manipolato attraverso due lenti di ingrandimento diverse, un puntatore a intero e un puntatore a carattere.

```c
#include <stdio.h>
int main(void) {
    int n = 1025;
    int *ip;
    char *cp;
    ip = &n;
    printf("\nIntero  -> Indirizzo: %p, Valore: %d", ip, *ip);
    printf("\nIntero+1-> Indirizzo: %p, Valore: %d", ip + 1, *(ip + 1));
    cp = (char *)ip; // cast
    printf("\nChar    -> Indirizzo: %p, Valore: %d", cp, *cp);
    printf("\nChar+1  -> Indirizzo: %p, Valore: %d\n", cp + 1, *(cp + 1));
    return 0;
}
```

i Analisi della memoria: Cosa accade "sotto il cofano"?

Il numero intero `1025` corrisponde alla somma $1024 + 1$. In formato binario a 32 bit (4 byte), questa cifra è rappresentata in memoria in questo modo:
`00000000 00000000 00000100 00000001`
Sulla quasi totalità delle architetture moderne (che utilizzano la convenzione *Little-Endian*), il byte meno significativo (quello all'estrema destra) viene salvato all'indirizzo di memoria più basso. I 4 byte che compongono il numero `1025` saranno quindi disposti in RAM così:

- Byte 0 (indirizzo base): `00000001` (Valore decimale: 1)
- Byte 1 (indirizzo + 1): `00000100` (Valore decimale: 4)
- Byte 2 (indirizzo + 2): `00000000` (Valore decimale: 0)
- Byte 3 (indirizzo + 3): `00000000` (Valore decimale: 0)

Quando stampiamo `*ip`, il compilatore sa che è un `int *`: parte dall'indirizzo base e **legge tutti e 4 i byte insieme**, ricostruendo il numero `1025`.
Quando eseguiamo un *cast* e assegnamo l'indirizzo a `cp` (che è un `char *`), cambiamo le regole di lettura. Stampando `*cp`, il compilatore **legge solo il primo byte** all'indirizzo base, restituendo il valore decimale `1`. Se eseguiamo l'aritmetica `*(cp + 1)`, il puntatore si sposta in avanti di **un solo byte**, e legge esclusivamente il secondo byte, restituendo il valore decimale `4`.

11 Livello 9

11.1 Introduzione all'allocazione dinamica

Fino a questo punto, la gestione della memoria è stata delegata interamente al compilatore attraverso l'allocazione **statica**. In questo scenario, le variabili locali vengono allocate nell'area di memoria denominata **Stack** all'atto della chiamata di una funzione, vengono distrutte (deallocate) in modo automatico al termine della stessa e la quantità di memoria necessaria deve essere nota a tempo di compilazione (*compile-time*).

Per superare quest'ultimo vincolo, il C offre gli strumenti per l'allocazione **dinamica** della memoria. Questo paradigma trasferisce il controllo dal compilatore al programmatore, consentendo di richiedere spazio durante l'esecuzione (*run-time*) all'interno di un'ampia area di memoria denominata **Heap**.

11.1.1 `malloc` e `free`

La gestione manuale dell'Heap si basa sulle funzioni messe a disposizione dalla libreria di sistema `<stdlib.h>`. L'operazione di allocazione si esegue (tipicamente) tramite la funzione `malloc()` (*memory allocation*) che richiede in ingresso la dimensione esatta in byte dello spazio desiderato e restituisce l'indirizzo del blocco allocato.

Si consideri la seguente istruzione fondamentale:

```
p = malloc(sizeof(int));
```

In questa riga si concentrano diverse operazioni cruciali:

1. L'operatore `sizeof(int)` calcola la dimensione necessaria per ospitare un numero intero.
2. La funzione `malloc` riserva quello spazio nell'Heap.
3. L'indirizzo di memoria viene infine assegnato al puntatore `p` (il quale risiede invece nello Stack).

Da questo momento, l'intero allocato nell'Heap è accessibile esclusivamente per via indiretta dereferenziando il puntatore (es. `*p = 10;`).

A differenza dello Stack, l'Heap non è soggetto a pulizia automatica, una volta che la variabile dinamica non è più necessaria, lo spazio deve essere liberato tramite la funzione `free(p);`.

11.2 Le liste concatenate

Una **lista concatenata** è una struttura dati composta da un insieme di elementi omogenei (tutti del medesimo tipo: una `struct` definita dall'utente), la cui implementazione si basa sull'utilizzo dei puntatori. In questa sezione verrà illustrato come creare e gestire una lista concatenata in cui gli elementi non sono dichiarati staticamente a priori, bensì generati dinamicamente durante l'esecuzione del programma.

Vi sono alcuni principi fondamentali e imprescindibili da tenere a mente quando si opera con le liste concatenate:

- **Allocazione dinamica:** ogni qualvolta sia necessario creare un nuovo nodo (elemento), occorre allocare esplicitamente lo spazio in memoria. Questa operazione si esegue avvalendosi della funzione `malloc` (inclusa nella libreria `<stdlib.h>`). La `malloc` restituisce un puntatore all'area allocata e richiede come parametro la dimensione in byte dell'elemento. Un esempio di chiamata per una struttura definita dall'utente denominata `mia_lista` è: `malloc(sizeof(struct mia_lista))`. Si osservi il seguente frammento utilizzato per allocare la memoria nell'heap necessaria a un singolo elemento:

```
struct mia_lista *p;
p = malloc(sizeof(struct mia_lista));
```

A questo punto, è possibile utilizzare il puntatore `p` per accedere alla nuova locazione di memoria appena allocata.

- **Puntatore interno (concatenazione):** ogni singolo elemento della lista sarà collegato al nodo successivo tramite un puntatore interno. Di conseguenza, in fase di definizione della `struct`, è tassativo includere un puntatore (come campo membro) al medesimo tipo di struttura (creando una struttura *autoreferenziale*):

```
struct mia_lista {
    int num;
    struct mia_lista *next;
};
```

L'omissione di questo puntatore (denominato `next` nell'esempio) renderà impossibile la concatenazione dei nodi.

- **Terminazione della lista:** per utilizzare e scansionare correttamente la lista, è indispensabile garantire che il puntatore interno dell'ultimo elemento della lista sia esplicitamente impostato a `NULL`. Questo costituisce il criterio di arresto che permette di iterare sui nodi fermandosi nel momento esatto in cui la lista è terminata.
- **Puntatore di testa (Head):** dove si trova fisicamente la lista? L'intera lista è identificata da un singolo puntatore rivolto al suo primo elemento. Se i nodi sono concatenati correttamente e l'ultimo punta a `NULL`, è sufficiente questo singolo puntatore esterno per mantenere le coordinate di partenza. In assenza di tale riferimento, non sarà più possibile accedere alla lista e deallocare (tramite `free`) la memoria occupata, generando un *memory leak* permanente per tutta la durata dell'esecuzione del programma.

Questi costituiscono i fondamenti teorici per la gestione di una lista. Tuttavia, la piena comprensione di queste dinamiche si acquisisce unicamente attraverso la pratica.

11.2.1 Crea e visualizza una lista

Vediamo un esempio dedicato alla gestione di una lista concatenata di interi. Il codice illustra la **creazione** dinamica (con inserimento in coda), la **scansione** per la stampa e, infine, la **deallocazione** della memoria per prevenire *memory leak*.

Il programma inizia con la definizione della struttura `lista` e la dichiarazione dei prototipi delle funzioni. Nel blocco del `main` si dichiara il puntatore di testa (`punt_lista`) e si inseriscono le chiamate alle funzioni, verificando che l'allocazione iniziale sia andata a buon fine.

```c
#include <stdio.h>
#include <stdlib.h>
struct lista {
    int num;
    struct lista *next;
};
struct lista *crealista();
void visualizza_lista(struct lista *);
void libera_lista(struct lista *);

int main(void) {
    struct lista *punt_lista;
    punt_lista = crealista();
    if(punt_lista != NULL) {
        visualizza_lista(punt_lista);
        libera_lista(punt_lista);
    }
    return 0;
}
```

La funzione `crealista` deve istanziare i nodi e concatenarli opportunamente attraverso l'uso di due puntatori:

- `p`: funge da riferimento alla "testa" della lista. È l'indirizzo che verrà restituito al `main` al termine dell'esecuzione.
- `paux` (puntatore ausiliario): partendo dalla testa, si sposta progressivamente in avanti per agganciare i nuovi nodi in coda.

```c
struct lista *crealista() {
    int n, i;
    struct lista *p, *paux;
    printf("Quanti elementi vuoi inserire? ");
    scanf("%d", &n);
    if(n < 1) p = NULL;
    else {
        // Allocazione e inizializzazione del nodo di testa
        p = malloc(sizeof(struct lista));
        printf("Inserisci valore: ");
        scanf("%d", &p->num);
        paux = p; // Il puntatore ausiliario si allinea alla testa
        // Ciclo di creazione e concatenazione dei nodi successivi
```

```
        for(i = 2; i <= n; i++) {
            paux->next = malloc(sizeof(struct lista));
            paux = paux->next; // Avanzamento sul nuovo nodo
            printf("Inserisci valore: ");
            scanf("%d", &paux->num);
        }
        paux->next = NULL; // Terminazione della lista
    }
    return p;
}
```

Per scansionare la lista, la funzione iterativa sfrutta il passaggio dei parametri per valore tipico del linguaggio C. Il parametro formale `p` riceve una copia dell'indirizzo di testa. All'interno del ciclo `while`, l'istruzione `p = p->next` permette di "scorrere" la lista nodo per nodo fino a incontrare il terminatore `NULL`. Operando su una copia, il puntatore originario nel `main` rimane intatto.

```
void visualizza_lista(struct lista *p) {
    while(p != NULL) {
        printf("%d ", p->num);
        p = p->next;
    }
    printf("\n");
}
```

L'ultima fase affronta la distruzione della struttura dati. La liberazione della memoria richiede un'attenzione particolare all'ordine delle operazioni. Se si eseguisse la sequenza `free(p); p = p->next;`, il programma tenterebbe di leggere il campo `next` di un'area di memoria che è già stata liberata, scatenando un errore noto come *Use After Free*. Per evitare tale situazione, è d'obbligo l'impiego di un puntatore temporaneo.

```
void libera_lista(struct lista *p) {
    struct lista *paux;

    while(p != NULL) {
        paux = p; // 1. Salva l'indirizzo del nodo corrente
        p = p->next; // 2. Avanza al nodo successivo
        free(paux); // 3. Dealloca il nodo corrente, isolato
    }
}
```

11.2.2 Inserisci elementi in una lista

Rispetto alla creazione in blocco vista in precedenza, un approccio più flessibile prevede l'aggiunta dinamica e iterativa dei singoli nodi durante l'esecuzione del programma. Le due tipologie standard per questa operazione sono l'inserimento in **testa** (che inserisce il nuovo elemento come primo nodo della catena) e l'inserimento in **coda** (che lo aggancia come ultimo elemento).

Di seguito viene analizzato un programma che implementa l'inserimento in coda, permettendo all'utente di gestire la lista tramite un menu testuale.

Il `main` funge da gestore di tutte le funzioni. È fondamentale notare l'inizializzazione del puntatore di testa: `struct lista *punt_lista = NULL;`. Questa istruzione stabilisce in modo esplicito che, all'avvio del programma, la lista è vuota. L'interfaccia utente è governata da un ciclo `while` accoppiato a uno `switch-case`, che chiama le funzioni mantenendo aggiornato il puntatore principale.

```c
#include <stdio.h>
#include <stdlib.h>
struct lista {
    int num;
    struct lista *next;
};
struct lista *ins_coda(struct lista *);
void visualizza_lista(struct lista *);
int main(void) {
    struct lista *punt_lista = NULL; // Inizializzazione a lista vuota
    int sel = -1;
    while(sel != 0) {
        printf("\n1- Inserisci elemento\n");
        printf("2- Visualizza elementi\n");
        printf("0- Esci\n");
        printf("Scelta: ");
        scanf("%d", &sel);
        switch (sel) {
            case 1:
                // Il puntatore viene aggiornato con il valore restituito
                punt_lista = ins_coda(punt_lista);
                break;
            case 2:
                if(punt_lista != NULL) {
                    printf("\nLista elementi: ");
                    visualizza_lista(punt_lista);
                } else {
                    printf("\nLista vuota\n");
                }
                break;
            case 0:
                printf("\nProgramma terminato\n");
                break;
            default:
                printf("\nScelta non valida\n");
        }
    }
    return 0;
}
```

L'operazione di inserimento in coda richiede una gestione dei casi limite. La funzione `ins_coda` accetta come parametro l'indirizzo di testa della lista e restituisce il valore (che potrebbe essere mutato). L'algoritmo è organizzato in tre fasi:

1. **Allocazione:** si crea in memoria il nuovo nodo isolato (`p1`) e se ne popola il campo dati con l'input dell'utente.
2. **Gestione del caso base (lista vuota):** se il puntatore di testa ricevuto in ingresso è `NULL`, significa che la lista è ancora vuota. Il nuovo nodo diviene la testa della lista. Il suo campo `next` viene impostato a `NULL` e l'indirizzo di `p1` viene assegnato al puntatore principale `p`.
3. **Gestione del caso generale (lista popolata):** se la lista contiene già dei nodi, il nuovo elemento `p1` è destinato a diventarne la coda e di conseguenza, il suo campo `next` viene forzato a `NULL`. Successivamente, si utilizza un puntatore ausiliario (`paux`) per attraversare l'intera lista partendo dalla testa, fino a individuare l'attuale ultimo elemento (condizione: `while(paux->next != NULL)`). Trovato l'elemento terminale, se ne modifica il campo `next` per agganciarlo fisicamente al nuovo nodo `p1`.

```
struct lista *ins_coda(struct lista *p) {
    struct lista *p1, *paux;
    // 1. Allocazione e popolamento del nuovo nodo
    p1 = malloc(sizeof(struct lista));
    printf("Inserisci valore: ");
    scanf("%d", &p1->num);
    // 2. Caso Base: la lista è attualmente vuota
    if(p == NULL) {
        p = p1; // Il nuovo nodo diventa la testa
        p->next = NULL; // La lista termina immediatamente
    }
    // 3. Caso generale: la lista ha già degli elementi
    else {
        p1->next = NULL; // Il nuovo nodo va in coda
        paux = p; // Il puntatore ausiliario parte dalla testa
        // Scorrimento fino all'ultimo nodo esistente
        while(paux->next != NULL) {
            paux = paux->next;
        }
        // Aggancio del nuovo nodo in coda all'ultimo elemento
        paux->next = p1;
    }
    // Restituzione del puntatore di testa
    return p;
}
```

Architettura della Memoria: Stack vs Heap

Ogni programma C in esecuzione gestisce la memoria allocata secondo due paradigmi fondamentali: **allocazione statica** e **allocazione dinamica**.

- **Lo Stack (allocazione statica):** è una regione di memoria a dimensione fissa, gestita in modo **automatico** dal compilatore secondo una logica LIFO (Last-In, First-Out). Ospita le variabili locali e i parametri delle funzioni. Ha un tempo di accesso estremamente rapido, ma la sua rigidità dimensionale (definita a *compile-time*) espone al rischio di esaurimento dello spazio (*Stack Overflow*) se si tenta di allocare array statici troppo grandi.
- **L'Heap (allocazione dinamica):** è un'ampia area di memoria gestita

manualmente dal programmatore tramite funzioni specifiche (es. `malloc`, `free`). La sua dimensione è elastica e determinata a *run-time*, permettendo di gestire strutture dati che crescono o si riducono dinamicamente (come le liste concatenate). L'accesso ai dati (che avviene tramite puntatori) è leggermente meno rapido rispetto allo stack, ma offre una maggiore flessibilità.

11.3 Array dinamici

Sebbene nel linguaggio C non esista nativamente il concetto di array a dimensione dinamica, è possibile simularne perfettamente il comportamento richiedendo al sistema operativo l'allocazione di blocchi contigui di memoria tramite le funzioni `malloc` o `calloc`.

Si analizzi il seguente esempio applicativo:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int i, n;
    int *ptr;
    printf("Inserisci la dimensione dell'array: ");
    scanf("%d", &n);
    // Allocazione dinamica tramite calloc (azzera anche la memoria)
    ptr = calloc(n, sizeof(int));
    // Alternativa equivalente con malloc (non azzera la memoria):
    // ptr = (int*) malloc(n * sizeof(int));
    for(i = 0; i < n; i++) {
        printf("\nInserisci valore: ");
        // Acquisizione tramite aritmetica dei puntatori
        scanf("%d", ptr + i);
        // Equivalente alla notazione vettoriale: scanf("%d", &ptr[i]);
    }
    printf("\nArray inserito: ");
    for(i = 0; i < n; i++) {
        printf("%d ", *(ptr + i));
        // Equivalente alla notazione vettoriale: printf("%d ", ptr[i]);
    }
    // Passaggio fondamentale: rilascio della memoria allocata
    free(ptr);
    return 0;
}
```

11.4 Array dinamici e funzioni

L'integrazione tra l'allocazione dinamica e la programmazione modulare richiede una gestione attenta degli indirizzi di memoria. Quando si demanda a una funzione il compito di allocare e inizializzare un array dinamico, si pone il problema di come trasferire l'indirizzo

della memoria allocata al blocco chiamante (il `main`), affinché quest'ultimo possa utilizzarla (per visualizzarne i valori) o deallocarla.

Le due soluzioni proposte mostrano due paradigmi differenti per risolvere questo problema.

11.4.1 Soluzione 1: restituzione dell'indirizzo

```c
#include <stdio.h>
#include <stdlib.h>
int * allocate_array(int size);
int main (void) {
    int i,n, *ptr;
    printf("\nInserisci dimensione array: ");
    scanf("%d",&n);
    ptr=allocate_array(n);
    printf("ptr: %p\n",ptr);
    for(i=0;i<n;i++) printf("%d ",*(ptr+i));
    free(ptr);
}
int * allocate_array(int size) {
    int i,*p;
    p=calloc(size, sizeof(int));
    printf("p: %p\n",p);
    for(i=0;i<size;i++)
        scanf("%d",p+i);
    return p;
}
```

Il primo approccio è il più intuitivo. La funzione viene progettata per restituire esplicitamente un puntatore al tipo di dato allocato. All'interno della funzione viene dichiarato un puntatore locale `p` a cui si assegna l'indirizzo generato da `calloc`. Dopo aver popolato l'array acquisendo i dati dall'utente, la funzione termina con l'istruzione `return p;`. Nel `main`, l'indirizzo restituito viene semplicemente catturato tramite un'operazione di assegnamento: `ptr = allocate_array(n);`. Da questo momento, il puntatore `ptr` del `main` referenzia correttamente l'area Heap allocata dalla funzione.

11.4.2 Soluzione 2: doppio puntatore

```c
#include <stdio.h>
#include <stdlib.h>
void allocate_array(int size, int **p);
int main (void) {
    int i,n, *ptr;
    printf("Inserisci dimensione array: ");
    scanf("%d",&n);
    printf("Indirizzo ptr: %p\n",&ptr);
    allocate_array(n,&ptr);
```

```
    printf("Valore di ptr in main: %p\n",ptr);
    for(i=0;i<n;i++) printf("%d ",*(ptr+i));
    free(ptr);
}
void allocate_array(int size, int **p) { //p=&ptr;
    int i;
    printf("Indirizzo ptr nella f: %p\n",p);
    *p=calloc(size, sizeof(int));
    //*p equivale al valore di ptr del main -> ptr = calloc...
    printf("Contenuto di ptr in f: %p \n",*p);
    for(i=0;i<size;i++) scanf("%d",*p+i); //carico in array dinamico
}
```

Il secondo approccio è di grande rilevanza, in quanto elimina l'istruzione di `return` e la funzione non restituisce nulla (`void`), ma modifica direttamente il puntatore definito nel `main`. Per far sì che la funzione modifichi in modo permanente la variabile del modulo chiamante, è necessario passare l'indirizzo di tale variabile. Poiché la variabile da modificare nel `main` è già un puntatore (`int *ptr`), il suo indirizzo `&ptr` sarà inevitabilmente un **puntatore a puntatore** (`int **p`). All'interno della funzione, l'istruzione `*p = calloc(size, sizeof(int));` esegue una dereferenziazione, non sta assegnando la memoria al parametro locale `p`, ma sta scrivendo l'indirizzo restituito dalla `calloc` direttamente dentro la variabile `ptr` del `main`. Per scorrere e popolare l'array, si utilizza l'espressione `*p + i`, in questo modo si legge prima l'indirizzo base contenuto nel `main` (`*p`) e poi si applica l'aritmetica dei puntatori aggiungendo l'offset `i`.

12 Livello 10

12.1 La gestione dei file

L'introduzione dei file rappresenta uno snodo cruciale nella programmazione in C perché offre finalmente la possibilità di rendere **persistenti** i dati elaborati da un programma, superando il limite della volatilità della memoria RAM (Stack e Heap).

A differenza di linguaggi di più alto livello, il C non fornisce astrazioni sofisticate per la gestione dei file, ma si affida a un approccio a basso livello basato sui **puntatori**.

Un aspetto da valutare in fase di progettazione (che dipende dalla destinazione d'uso dei dati) è la scelta tra file di testo e file binari:

- **File di testo:** i dati vengono convertiti in sequenze di caratteri. Sono leggibili dall'utente tramite un normale editor di testo e risultano ideali per garantire la portabilità esterna (es. file di configurazione, file di log). L'acquisizione di questi dati richiede però un'operazione di *parsing*, ovvero la trasformazione della stringa di caratteri letta in dati strutturati (interi, float) utilizzabili dal programma.
- **File binari:** contengono l'esatta rappresentazione in byte presente nella memoria del calcolatore. Non sono leggibili da un essere umano, ma non richiedono alcuna conversione o *parsing*. Sono la scelta perfetta per salvare o caricare rapidamente intere strutture dati.

12.1.1 Apertura e chiusura

Per operare sui file è necessario includere la libreria standard `<stdio.h>`, la quale definisce il tipo derivato `FILE`. Ogni operazione di lettura o scrittura avviene tramite un puntatore a questa struttura.

```
FILE *fileptr; // Dichiarazione del puntatore al file
```

> **Nota**
>
> La struttura interna del tipo `FILE` non è strettamente standardizzata, ma dipende dall'implementazione specifica della libreria sul Sistema Operativo in uso (Windows, Linux, macOS). Il programmatore interagisce con essa solo tramite il puntatore.

12.1.1.1 L'apertura: `fopen()`

Prima di qualsiasi manipolazione, il file deve essere aperto tramite la funzione `fopen()`, che associa il puntatore al file fisico sul disco, specificando la modalità di accesso.

Modalità	Significato	Comportamento se il file non esiste	Comportamento se esiste
`r`	Solo lettura	Restituisce `NULL`	Apre il file dall'inizio
`w`	Solo scrittura	Crea un nuovo file	**Sovrascrive** il contenuto
`a`	Append (scrittura in coda)	Crea un nuovo file	Scrive alla fine del file
`r+`	Lettura e Scrittura	Restituisce `NULL`	Apre il file dall'inizio
`w+`	Scrittura e Lettura	Crea un nuovo file	**Sovrascrive** il contenuto
`a+`	Lettura e Scrittura in coda	Crea un nuovo file	Mantiene il contenuto, scrive in coda

Buona pratica: per i file binari, lo standard raccomanda di aggiungere il suffisso `b` alla modalità (es. `rb`, `wb`, `ab`). Sebbene su sistemi come Linux e macOS non vi sia differenza pratica, su sistemi Windows questa distinzione è indispensabile per evitare la corruzione dei dati dovuta alla traslazione automatica dei caratteri di fine riga.

12.1.1.2 La gestione degli errori e la chiusura: `fclose()`

È obbligatorio verificare sempre che l'apertura sia andata a buon fine controllando che il puntatore non sia `NULL`. In caso di errore, l'uso di `perror()` fornisce il dettaglio esatto dell'errore restituito dal Sistema Operativo. Infine, ogni file aperto deve essere chiuso con `fclose()`, operazione che garantisce il salvataggio fisico dei dati sul disco liberando il buffer.

```
fileptr = fopen("dati.txt", "r");
if (fileptr == NULL) {
    perror("Errore durante l'apertura del file");
    return 1; // Termina il programma con codice di errore
}
// ... operazioni sul file ...
fclose(fileptr); // Chiusura fondamentale
```

12.1.2 File di testo

La libreria standard offre tre famiglie di funzioni per gestire l'I/O sui file di testo, classificate in base alla granularità del dato trattato.

12.1.2.1 `fprintf()` e `fscanf()`

Agiscono in modo identico a `printf` e `scanf`, ma operano sul flusso di dati (stream) del file anziché sullo standard input/output.

La funzione `fscanf` restituisce il numero di elementi letti e convertiti con successo. Questo meccanismo è cruciale per leggere un file di testo fino alla sua conclusione logica (EOF - End Of File), anche quando non si conosce a priori il numero di elementi.

```c
// Esempio: Caricamento dinamico da file a un array
#include <stdio.h>
#define DIM 100
int main(void) {
    FILE *fileptr;
    int arr[DIM];
    int num, i = 0, n;
    fileptr = fopen("file_01.txt", "r");
    if(fileptr != NULL) {
        // Continua a leggere finché trova interi validi e non supera la DIM
        while(fscanf(fileptr, "%d", &num) == 1 && i < DIM) {
            arr[i] = num;
            i++;
        }
        fclose(fileptr);
        n = i; // Numero effettivo di elementi letti
    }
    return 0;
}
```

12.1.2.2 `fgets()` e `fputs()`

- `fgets(buffer, dimensione, fileptr)`: legge un'intera riga dal file fino al raggiungimento del carattere `\n` (incluso) o fino al limite della dimensione del buffer, aggiungendo automaticamente il terminatore di stringa `\0`.
- `fputs(stringa, fileptr)`: scrive una stringa sul file.

> **`fgets()` e la tastiera**
>
> Poiché la tastiera è considerata a tutti gli effetti un flusso di file di nome `stdin` (Standard Input), la funzione `fgets()` è raccomandata per l'acquisizione sicura di stringhe testuali dall'utente, superando i ben noti limiti della `scanf` (che si interrompe al primo spazio).

12.1.2.3 `fgetc()` e `fputc()`

Permettono di leggere e scrivere un carattere alla volta, utili per la manipolazione a bassissimo livello dei file di testo. Il ciclo di lettura tipico continua finché `fgetc` non restituisce la macro `EOF`.

12.1.3 File binari

I file binari bypassano le problematiche di formattazione. Le funzioni `fread()` e `fwrite()` consentono di spostare direttamente blocchi di memoria (low-level reading/writing) dalla RAM al disco e viceversa.

- `fread(destinazione, dimensione_elemento, numero_elementi, fileptr)`
- `fwrite(sorgente, dimensione_elemento, numero_elementi, fileptr)`

Questo approccio si rivela estremamente potente quando associato alle funzioni di riposizionamento spaziale del cursore.

12.1.3.1 `fseek()` e `ftell()`

L'accesso ai file non deve essere necessariamente sequenziale. Il C permette l'accesso "random" (casuale o diretto) spostando il cursore interno del file.

- **`fseek(fp, offset, origine)`**: sposta il cursore di un numero di byte pari a `offset` a partire da una specifica `origine`. Per indicare l'origine, lo standard C prescrive l'utilizzo delle seguenti macro (definite all'interno di `<stdio.h>`):
- `SEEK_SET`: inizio del file (sostituisce il valore numerico `0`)
- `SEEK_CUR`: posizione attuale del cursore (sostituisce il valore numerico `1`)
- `SEEK_END`: fine del file (sostituisce il valore numerico `2`)
- **`ftell(fp)`**: restituisce l'esatta posizione attuale del cursore espressa in byte.

> **!** Importante
>
> Sebbene molti compilatori tollerino ancora l'uso dei numeri diretti (`0`, `1`, `2`) per il parametro di origine, questa è considerata una cattiva pratica. L'uso rigoroso delle macro `SEEK_SET`, `SEEK_CUR` e `SEEK_END` garantisce la portabilità del codice su architetture diverse.