

LEZIONE 7

L'Astrazione del Codice e l'Anatomia dello Stack



Dispense di Programmazione 1 – Canale 0-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

L'Astrazione del Codice e l'Anatomia dello Stack

Nelle lezioni precedenti abbiamo imparato a strutturare programmi in grado di compiere scelte e ripetere istruzioni. Tuttavia, man mano che la complessità dei problemi aumenta, scrivere tutto il codice all'interno di un unico blocco monolitico (il `main`) porta inevitabilmente a codici confusi, difficili da leggere e pieni di duplicazioni (il pericoloso pattern del copia-incolla). Questa lezione introduce lo strumento fondamentale per superare questo limite: le **funzioni**. Impareremo a scomporre un problema complesso in sotto-problemi indipendenti, a definire l'interfaccia di una funzione e a comprendere cosa accade "sotto il cofano" della memoria RAM, analizzando la struttura dello **Stack** e l'evoluzione dei **Record di Attivazione**.

1.1 Il Limite del Codice Monolitico e il Principio DRY

1.1.1 Perché il copia-incolla è il male

Immaginiamo di dover trovare il valore massimo all'interno di tre array distinti nel nostro programma. Senza uno strumento per riutilizzare la logica di calcolo, l'unica soluzione possibile sarebbe duplicare lo stesso codice per tre volte, adattando manualmente i nomi delle variabili:

```
1 // Ricerca massimo nel primo array
2 int max1 = array1[0];
3 for (int i = 1; i < N; i++) {
4     if (array1[i] > max1) {
5         max1 = array1[i];
6     }
7 }
8
9 // Ricerca massimo nel secondo array (COPIA-INCOLLA)
10 int max2 = array2[0];
11 for (int i = 1; i < N; i++) {
12     if (array2[i] > max2) {
13         max2 = array2[i];
14     }
15 }
16
```

```
17 // E cos? via per il terzo array...
```

Quando ci troviamo a dover eseguire lo stesso calcolo su insiemi di dati diversi, la tentazione immediata è quella di copiare e incollare il blocco di codice interessato, modificando solo i nomi delle variabili. Questo approccio viola il principio cardine dell'Ingegneria del Software noto come **DRY (Don't Repeat Yourself)**.

Se in un blocco di codice duplicato è presente un bug (ad esempio un errore off-by-one nel limite di un ciclo), saremo costretti a rintracciare e correggere l'errore in ogni singola copia. Qualora dimenticassimo anche solo una di queste copie, il programma presenterebbe comportamenti inconsistenti.

1.1.2 L'approccio procedurale

Per risolvere questo problema, il linguaggio C adotta il paradigma **procedurale**. Esso permette di isolare una porzione di codice, assegnarle un nome auto-esplicativo e richiamarla ogni volta che ne sorge la necessità.

Questa operazione prende il nome di **astrazione**: l'utente della funzione ha bisogno di conoscere solo l'interfaccia (cosa entra e cosa esce) senza doversi preoccupare dei dettagli implementativi interni.

In realtà, abbiamo utilizzato le funzioni fin dal nostro primissimo programma: in C non è possibile scrivere codice senza ricorrere ad esse, dato che il `main` stesso è una funzione obbligatoria che funge da punto d'ingresso del programma.

Inoltre, abbiamo fatto largo uso delle funzioni della libreria standard (quali `printf`, `scanf`, `pow`, `sqrt`, `rand`, `srand`). Questo ci ha permesso di risparmiare tempo prezioso, di modularizzare il nostro codice e di appoggiarci a implementazioni robuste, sicure e ampiamente testate. Questo rappresenta già un enorme guadagno in termini di produttività.

Ma cosa succede se il nostro programma necessita di un calcolo specifico che non è presente in nessuna libreria di sistema? In questo caso, il C ci permette di costruire le nostre funzioni personalizzate (user-defined functions).

Per comprendere l'utilità pratica, consideriamo cosa dovremmo fare se volessimo calcolare la radice quadrata di un numero reale senza poter richiamare la funzione `sqrt` di `<math.h>`:

```
1 // Come calcoliamo la radice oggi grazie alle librerie (1 sola riga, zero
  ↪ pensieri):
2 double radice = sqrt(25.0);
3
4 // Come dovremmo scriverlo nel main se NON potessimo usare le funzioni:
5 double n = 25.0;
6 double radice;
7 if (n < 0) {
8     printf("Errore: radice di numero negativo!\n");
9     // gestione dell'errore...
10 } else {
11     // Algoritmo di approssimazione di Newton-Raphson iterativo
12     double x = n;
13     double y = 1.0;
14     double e = 0.000001;
15     while (x - y > e) {
16         x = (x + y) / 2;
```

```

17     y = n / x;
18 }
19 radice = x;
20 }
21
22 // E dovremmo fare questo COPIA-INCOLLA ogni volta che ci serve una
    ↪ radice nel programma!

```

1.2 Anatomia di una Funzione in C

Il concetto di funzione in informatica deriva direttamente dalla matematica. In matematica, una funzione f associa ad ogni elemento di un insieme di partenza (detto **dominio** D) uno e un solo elemento di un insieme di arrivo (detto **codominio** C), formalizzato come $f : D \rightarrow C$.

In programmazione facciamo esattamente lo stesso:

- Il **dominio** D è rappresentato dai tipi dei parametri formali accettati in ingresso.
- Il **codominio** C è rappresentato dal tipo di ritorno della funzione.
- La legge di associazione è descritta dal corpo della funzione che elabora gli input per produrre l'output.

Vediamo il parallelo sintattico:

Una funzione in C è definita da tre elementi essenziali che costituiscono la sua **firma** (o prototipo), seguiti dal **corpo** racchiuso tra parentesi graffe:

```

1 tipo_ritorno nome_funzione(tipo_parametro1 parametro1, tipo_parametro2
    ↪ parametro2, ...) {
2     // Corpo della funzione
3     return valore_ritorno; //deve essere di tipo "tipo_ritorno"
4 }

```

1. **Tipo di Ritorno:** Specifica il tipo di dato che la funzione restituisce al termine del suo calcolo (es. `int`, `double`, `char`). Se la funzione esegue solo delle azioni (come stampare a schermo) e non deve restituire alcun valore, si utilizza la parola chiave `void`.
2. **Nome Identificativo:** Segue le stesse regole dei nomi di variabili (deve iniziare con una lettera o underscore e non contenere spazi).
3. **Parametri Formali:** Le variabili dichiarate tra parentesi tonde che accolgono i dati in ingresso. Fungono da variabili locali temporanee che verranno popolate all'atto della chiamata.
4. **Istruzione return:** Termina immediatamente l'esecuzione della funzione e restituisce il valore specificato al chiamante. Se la funzione è di tipo `void` (non restituisce nulla), l'uso di `return`; è facoltativo e serve solo ad interrompere anticipatamente l'esecuzione del corpo. Se invece la funzione ha un tipo di ritorno non `void`, l'istruzione `return` con un valore coerente è obbligatoria per trasmettere il risultato al chiamante.

Facciamo un esempio pratico:

```
1 // Prototipo o Firma (Il "Cosa")
2 int calcola_somma(int a, int b);
3
4 // Definizione (Il "Come")
5 int calcola_somma(int a, int b) {
6     int risultato = a + b;
7     return risultato;
8 }
```

1.2.1 Prototipi e Definizioni

Poiché il compilatore C analizza il codice dall'alto verso il basso, se una funzione viene chiamata nel `main` prima di essere definita, il compilatore genererà un errore. Per evitare questo, si separano i **prototipi** (dichiarazioni) posti in cima al file, dalle **definizioni** effettive collocate in fondo:

Nota

Nomi dei parametri nel prototipo Nel prototipo di una funzione, l'inserimento dei nomi dei parametri formali è del tutto facoltativo. Ad esempio, dichiarare `double half(int);` è sintatticamente valido quanto `double half(int x);`. Il compilatore, in questa fase, necessita solo di conoscere la firma dei tipi (il tipo di ritorno e i tipi degli input). D'altronde, a livello assembly non esistono nomi di variabili, ma solo indirizzi, registri o offset in memoria.

```
1 #include <stdio.h>
2
3 // Dichiarazione (Prototipo)
4 double half(int x);
5
6 int main(void) {
7     int val = 5;
8     double ris = half(val); // Chiamata
9     printf("La meta' e': %f\n", ris);
10    return 0;
11 }
12
13 // Definizione
14 double half(int x) {
15     return (double)x / 2;
16 }
```

1.3 Il Flusso di Esecuzione e lo Scope delle Variabili

1.3.1 Alterazione del flusso

Quando viene effettuata una chiamata a funzione, il flusso principale (es. `main`) viene temporaneamente “congelato”. La CPU esegue un salto in memoria passando al corpo della funzione chiamata. Solo quando la funzione incontra l'istruzione `return` (o termina il suo blocco), il controllo ritorna al chiamante che si “scongela” e riprende l'esecuzione dall'istruzione immediatamente successiva.

Consideriamo il seguente esempio di codice numerato:

```
1
2 1.  int calcola_somma(int a, int b) {
3
4 2.      int risultato = a + b;
5
6 3.      return risultato;
7
8 4.  }
9
10 5.
11
12 6.  int main(void) {
13
14 7.      int x = 5, y = 3;
15
16 8.      int totale = calcola_somma(x, y);
17
18 9.      printf("Totale: %d\n", totale);
19
20 10.     return 0;
21
22 11. }
```

L'ordine di esecuzione delle righe è il seguente:

1. **Riga 6-7:** Il programma inizia dal `main`. Vengono allocate e inizializzate le variabili locali `x` (5) e `y` (3).
2. **Riga 8:** Viene invocata la funzione `calcola\_somma`. Il flusso principale si interrompe: i valori di `x` e `y` vengono copiati rispettivamente in `a` e `b`. La CPU esegue un salto in memoria.
3. **Riga 1:** Inizia l'esecuzione di `calcola\_somma`. I parametri formali `a` (5) e `b` (3) sono ora pronti in memoria.
4. **Riga 2:** Viene dichiarata la variabile locale `risultato` ed eseguito il calcolo ($5 + 3 = 8$).
5. **Riga 3:** L'istruzione `return risultato` restituisce il valore 8. Lo Stack Frame di `calcola\_somma` viene distrutto.
6. **Riga 8 (ripresa):** Il controllo ritorna al `main`. Il valore restituito (8) viene assegnato alla variabile `totale`.
7. **Riga 9:** Viene stampato il testo "Totale: 8".
8. **Riga 10:** Il `main` termina restituendo 0 al Sistema Operativo.

Suggerimento

Ispezione live con il Debugger Possiamo osservare fisicamente questa alterazione del flusso utilizzando un debugger (come LLDB in VS Code):

- **Step Over (F10)**: Esegue la riga corrente senza entrare nelle funzioni (se applicato alla riga 8, salta direttamente alla 9 calcolando il risultato).
- **Step Into (F11)**: Entra all'interno della funzione chiamata (se premuto alla riga 8, sposta il cursore di esecuzione alla riga 1).
- **Step Out (Shift+F11)**: Completa l'esecuzione della funzione corrente e ritorna immediatamente al chiamante.

1.3.2 Passaggio per Valore (Copia)

In C, il passaggio dei parametri avviene **sempre per valore**. Ciò significa che quando passiamo una variabile come argomento (parametro attuale) a una funzione, il sistema non passa la variabile stessa, ma una **copia del suo valore** che viene memorizzata nel parametro formale.

Di conseguenza, qualsiasi modifica apportata al parametro formale all'interno della funzione **non ha alcun effetto** sulla variabile originale del chiamante.

Attenzione

Parametri Formali vs Parametri Attuali

- **Parametri Formali**: Le variabili dichiarate nella firma della funzione che definiscono l'interfaccia (es. `int a`, `int b` in `calcola\{}_somma`). Esistono solo all'interno del corpo della funzione.
- **Parametri Attuali (o Argomenti)**: I valori concreti o le variabili passati al momento della chiamata (es. `x`, `y` o `5`, `3` nel `main`).

Vediamo un esempio di codice che dimostra che le modifiche ai parametri formali non influenzano i parametri attuali:

```
1  #include <stdio.h>
2
3  void incrementa(int n) {
4      n = n + 1; // Modifica solo la copia locale 'n'
5      printf("Valore dentro la funzione: %d\n", n);
6  }
7
8  int main(void) {
9      int x = 10;
10     incrementa(x); // Passiamo x (parametro attuale) alla funzione
11     printf("Valore nel main dopo la chiamata: %d\n", x); // x rimarrà 10!
12     return 0;
13 }
```

1.3.3 Scope Locale (Block Scope)

Le variabili dichiarate all'interno di una funzione (compresi i parametri formali) hanno uno scope locale: nascono al momento della chiamata e vengono distrutte non appena la funzione termina. Variabili dichiarate in funzioni diverse possono avere lo stesso nome senza causare conflitti, poiché risiedono in aree di memoria differenti.

```

1  #include <stdio.h>
2
3  void funzioneA(void) {
4      int valore = 42; // Locale a funzioneA
5      printf("Dentro funzioneA: %d\n", valore);
6  }
7
8  void funzioneB(void) {
9      int valore = 99; // Locale a funzioneB. Non collide con quella di
    ↪ funzioneA!
10     printf("Dentro funzioneB: %d\n", valore);
11 }
12
13 int main(void) {
14     funzioneA();
15     funzioneB();
16     // printf("%d", valore); // ERRORE DI COMPILAZIONE! 'valore' non
    ↪ esiste nel main
17     return 0;
18 }

```

Le **variabili globali** (dichiarate fuori da qualsiasi funzione) sono invece accessibili da chiunque. Il loro uso è vivamente sconsigliato poiché rende il comportamento del programma non deterministico e difficile da debuggare.

```

1  #include <stdio.h>
2
3  int contatore = 0; // VARIABILE GLOBALE - Sconsigliata!
4
5  void incrementa_contatore(void) {
6      contatore++; // Modifica lo stato globale
7  }
8
9  void resetta_contatore(void) {
10     contatore = 0; // Modifica lo stato globale inaspettatamente
11 }
12
13 int main(void) {
14     incrementa_contatore();
15     incrementa_contatore();
16     printf("Contatore: %d\n", contatore); // Stampa 2
17
18     resetta_contatore(); // Se resetta_contatore viene chiamata altrove,
    ↪ lo stato si corrompe
19
20     printf("Contatore dopo reset: %d\n", contatore); // Stampa 0!
21     return 0;
22 }

```

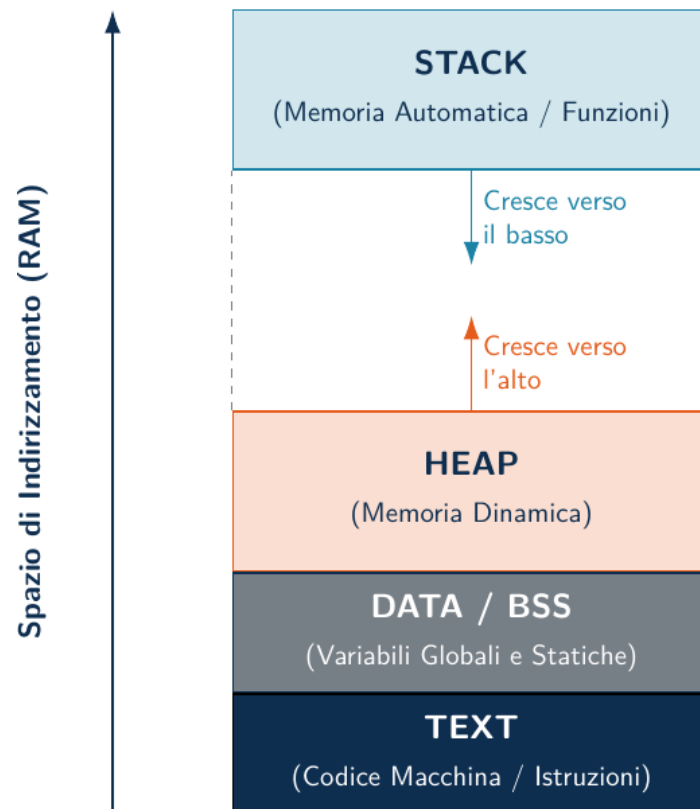


Figura 1.1: Layout Memoria RAM

1.4 Sotto il Cofano: Stack e Record di Attivazione

1.4.1 Organizzazione della RAM

La memoria RAM allocata per un programma in esecuzione è suddivisa in quattro aree principali:

- **Text:** Contiene il codice macchina (le istruzioni CPU).
- **Data / BSS:** Ospita le variabili globali e statiche. Il segmento **Data** memorizza le variabili inizializzate esplicitamente (es. `int x = 42;`), mentre il segmento **BSS** (*Block Started by Symbol*) memorizza quelle non inizializzate (es. `int y;`), che il Sistema Operativo azzerava automaticamente all'avvio.
- **Heap:** Destinata all'allocazione dinamica manuale.
- **Stack:** Struttura LIFO (Last In, First Out) gestita automaticamente per le funzioni.

1.4.2 Lo Stack Frame (Record di Attivazione)

Ogni volta che una funzione viene invocata, viene riservato nello Stack un blocco contiguo di memoria chiamato **Stack Frame** (o Record di Attivazione). Questo frame contiene:

1. I parametri passati alla funzione.
2. L'indirizzo di ritorno (l'istruzione del chiamante a cui saltare alla fine).
3. Il puntatore al vecchio frame (per ripristinare lo stato precedente).
4. Le variabili locali della funzione.

Quando la funzione termina (arriva al `return`), il suo Stack Frame viene rimosso dallo Stack (operazione di `Pop`) e lo spazio di memoria viene immediatamente invalidato, rendendolo disponibile per le chiamate successive. Questo spiega perché le variabili locali non sopravvivono al termine della funzione.

1.4.3 Allineamento e Padding a 16 Byte

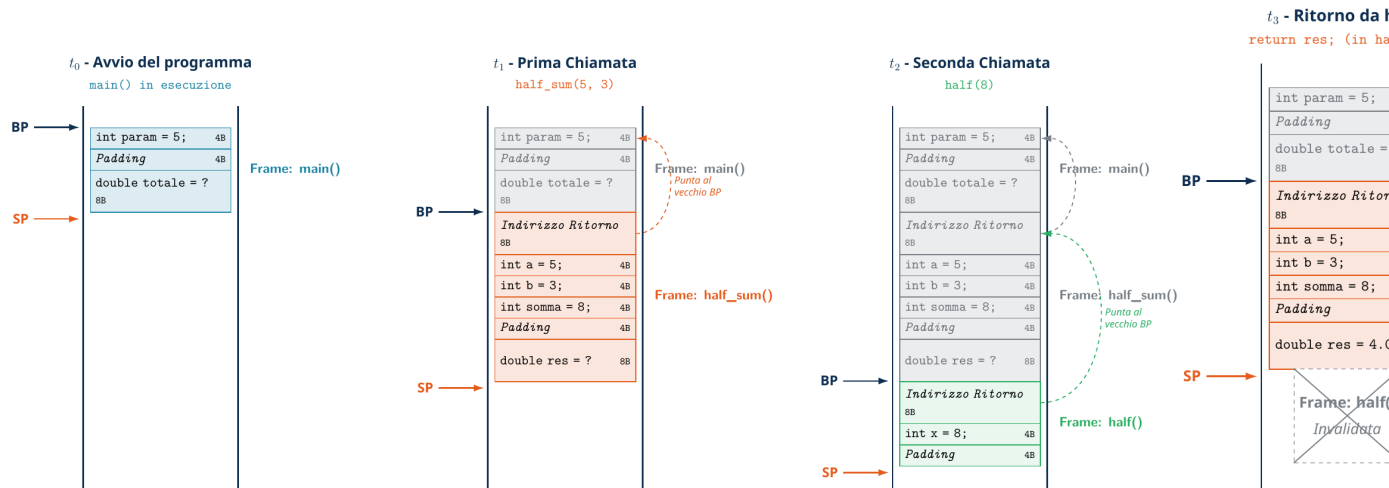
Nelle moderne architetture hardware a 64-bit, il trasferimento dei dati tra RAM e CPU avviene tramite bus ottimizzati per blocchi di grandi dimensioni. Per questioni di efficienza prestazionale, il compilatore richiede che ogni Stack Frame sia allineato a multipli di **16 byte**.

Se la somma dei byte occupati dalle variabili locali e dai parametri non corrisponde esattamente a un multiplo di 16, il compilatore inserisce dei byte vuoti invisibili chiamati **padding** per completare l'allineamento. Ad esempio:

- Se dichiariamo una sola variabile locale `int` (4 byte), lo Stack Frame verrà comunque allineato a 16 byte inserendo 12 byte di padding.

Consideriamo il codice d'esempio corrispondente all'evoluzione delle immagini sottostanti:

```
1  double half(int x) {
2      return (double)x / 2;
3  }
4
5  int half_sum(int a, int b) {
6      int somma = a + b;
7      double res = half(somma);
8      return res;
9  }
10
11 int main(void) {
12     int param = 5;
13     double totale = half_sum(param, 3);
14     return 0;
15 }
```



Attenzione

Come fa la CPU a sapere quanta memoria serve? Come fa il sistema a stabilire in anticipo la dimensione corretta di ogni Stack Frame? La risposta sta nel lavoro svolto dal **compilatore** a tempo di compilazione. Grazie al **sistema dei tipi** del C, il compilatore conosce a priori la dimensione in byte di ogni variabile locale e parametro dichiarato (es. `int` occupa 4 byte, `double` occupa 8 byte). Analizzando il corpo di ciascuna funzione, il compilatore somma lo spazio occupato dalle variabili, calcola l'allineamento a 16 byte (aggiungendo il padding necessario) e genera le istruzioni macchina appropriate per spostare lo Stack Pointer (es. `sub rsp, 24`).

1.4.4 I Registri RSP e RBP

La CPU tiene traccia del frame attivo attraverso due registri fondamentali:

- **RSP (Stack Pointer)**: Punta costantemente alla cima dello Stack (l'indirizzo più basso correntemente occupato). Si sposta continuamente ad ogni push o pop.
- **RBP (Base Pointer)**: Punta a un indirizzo fisso che fa da base per lo Stack Frame corrente. Tutte le variabili locali e i parametri all'interno della funzione vengono referenziati calcolando un offset rispetto ad RBP (es. `RBP - 4`).

Vediamo un semplicissimo esempio di codice C e il corrispondente codice assembly (in sintassi Intel x86-64) commentato per capire il ruolo dei registri e dello Stack:

```

1 // Codice C
2 int somma(int a, int b) {
3     return a + b;
4 }
5
6 int main(void) {
7     int ris = somma(2, 3);
8     return 0;
9 }

```

```

1 ; Assembly Intel x86-64 corrispondente
2 main:
3     ; 1. Preparazione dei parametri negli appositi registri
4     mov edi, 2      ; Metti il primo argomento (2) nel registro EDI
5     mov esi, 3      ; Metti il secondo argomento (3) nel registro ESI
6
7     ; 2. Invocazione della funzione
8     call somma      ; Salva l'indirizzo della prossima istruzione nello
    ↪ Stack
9
10                    ; e salta all'etichetta "somma"
11
12    ; ... La CPU ritorna esattamente qui al termine della funzione ...
13    mov DWORD PTR [rbp-4], eax ; Salva il valore restituito in EAX
    ↪ dentro 'ris'
14    ret
15
16 somma:
17    ; 3. Setup (Prologo) del nuovo Stack Frame
18    push rbp         ; Salva il Base Pointer del chiamante nello Stack
19    mov rbp, rsp     ; Imposta il nuovo Base Pointer pari allo Stack
    ↪ Pointer corrente
20
21    ; 4. Esecuzione del calcolo
22    add edi, esi     ; Somma a + b (EDI + ESI)
23    mov eax, edi     ; Memorizza il risultato nel registro EAX (registro
    ↪ di ritorno)
24
25    ; 5. Distruzione (Epilogo) dello Stack Frame
26    pop rbp         ; Ripristina il vecchio Base Pointer del chiamante
    ↪ dallo Stack
    ret              ; Preleva l'indirizzo di ritorno dallo Stack e vi
    ↪ salta indietro

```

Suggerimento

Verifica con Python Tutor Per visualizzare dinamicamente l'evoluzione dello Stack ad ogni istruzione (le chiamate di funzioni, il ciclo di vita delle variabili locali e il ritorno al chiamante), puoi copiare il codice C d'esempio su pythontutor.com impostando l'ambiente di visualizzazione in C (GCC). Potrai eseguire il codice passo-passo simulando graficamente la pila dei record di attivazione!

1.5 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Quale tipo di ritorno si utilizza per dichiarare una funzione che non restituisce alcun valore?

- A) int
- B) void
- C) NULL
- D) double

Soluzione
Risposta corretta: B
Spiegazione: La parola chiave void indica esplicitamente al compilatore che la funzione esegue solo istruzioni (effetti collaterali come stampe) e non produce un valore da restituire al chiamante.

Quiz 2

Domanda

2. Cosa accade a una variabile passata come argomento a una funzione se viene modificata all'interno della funzione stessa?

- A) Viene modificata anche nel chiamante
- B) Il programma va in errore di segmentazione
- C) Rimane invariata nel chiamante poiché viene passata solo una copia
- D) Diventa una variabile globale

Soluzione
Risposta corretta: C
Spiegazione: Il C adotta rigorosamente il passaggio dei parametri per valore. La funzione riceve una copia fisica della variabile originale posizionata in un indirizzo di memoria differente (il proprio Stack Frame).

Quiz 3

Domanda

3. Qual è lo scopo del prototipo di una funzione (Forward Declaration)?

- A) Allocare la memoria nello Stack
- B) Informare il compilatore dell'esistenza della funzione prima del suo utilizzo effettivo nel codice
- C) Ottimizzare l'esecuzione del codice assembly
- D) Dichiarare variabili globali

Spiegazione: Il prototipo dice al compilatore il nome, il tipo di ritorno e i parametri della funzione in modo che possa validare sintatticamente le chiamate prima di averne letto l'effettiva definizione del corpo.

Risposta corretta: B

Soluzione

Quiz 4

Domanda

4. In quale segmento di memoria RAM vengono allocati gli Stack Frame?

- A) Text Segment
- B) Heap
- C) Stack
- D) Data/BSS

Spiegazione: Lo Stack è l'area di memoria automatica e organizzata a pila adibita proprio alla gestione delle chiamate a funzione e delle loro variabili locali.

Risposta corretta: C

Soluzione

Quiz 5

Domanda

5. Cosa si intende per "Padding" all'interno di uno Stack Frame?

- A) L'eliminazione delle variabili locali inutilizzate
- B) L'inserimento di byte vuoti per allineare il frame a blocchi di 16 byte
- C) Il passaggio dei parametri tramite registri CPU
- D) Un errore di overflow dello Stack

Spiegazione: Nelle architetture a 64-bit il compilatore esegue il padding per allineare lo Stack Frame ai confini di 16 byte per massimizzare la velocità di trasferimento del bus hardware.

Risposta corretta: B

Soluzione

Quiz 6

Domanda

6. Quale registro punta costantemente alla base stabile del record di attivazione

corrente?

- A) RSP
- B) RAX
- C) RBP
- D) RIP

Soluzione
Risposta corretta: C
Spiegazione: Il registry RBP (Base Pointer) rimane fisso per tutta la durata della funzione ed è usato come riferimento costante per calcolare gli indirizzi delle variabili locali.

Quiz 7

Domanda

7. Cosa succede alla memoria allocata per le variabili locali di una funzione quando questa termina con un return?

- A) Viene spostata nell'Heap
- B) Viene deallocata automaticamente rimuovendo lo Stack Frame (Pop)
- C) Rimane memorizzata per sempre
- D) Viene salvata su file

Soluzione
Risposta corretta: B
Spiegazione: Al termine della funzione, lo Stack Pointer RSP viene ripristinato, invalidando lo Stack Frame associato. La memoria occupata diventa immediatamente riutilizzabile per altre chiamate.

Quiz 8

Domanda

8. Qual è il comportamento se una funzione dichiara una variabile locale con lo stesso nome di una variabile del main?

- A) Il compilatore genera un errore di duplicazione
- B) Le due variabili condividono lo stesso spazio di memoria
- C) La variabile locale nasconde quella del main senza conflitti (sono in due Stack Frame separati)
- D) Il programma va in crash all'avvio

Soluzione**Risposta corretta: C**

Spiegazione: Ciascuna funzione lavora esclusivamente all'interno del proprio scope locale. Le variabili risiedono in indirizzi di memoria fisicamente distinti all'interno dei rispettivi record di attivazione.

Quiz 9**Domanda**

9. Perché l'uso delle variabili globali è considerato una pessima pratica di programmazione?

- A) Rende l'esecuzione del programma più lenta
- B) Impedisce l'uso dei cicli `for`
- C) Rompe l'incapsulamento rendendo lo stato del programma imprevedibile e difficile da debuggare
- D) Funzionano solo con il tipo di dato `char`

Soluzione**Risposta corretta: C**

Spiegazione: Qualsiasi funzione può modificare una variabile globale inaspettatamente, creando effetti collaterali difficili da tracciare e compromettendo l'indipendenza delle singole procedure.

Quiz 10**Domanda**

10. In che direzione cresce lo Stack nella memoria RAM nella convenzione comune?

- A) Verso indirizzi di memoria più alti
- B) Rimane di dimensione fissa senza crescere
- C) Verso indirizzi di memoria più bassi
- D) Cresce in modo casuale ad ogni esecuzione

Soluzione**Risposta corretta: C**

Spiegazione: Nella convenzione architetturale standard, lo Stack cresce dall'alto verso il basso (ovvero verso indirizzi numericamente decrescenti) per evitare di scontrarsi con l'Heap che cresce dal basso verso l'alto.

1.6 Esercizi Pratici

Esercizio 1: Il Cubo di un intero

Domanda

Scrivete una funzione `int cubo(int x)` che accetti un numero intero e ne restituisca il cubo. Scrivete un `main` che legga un numero da tastiera, chiami la funzione e stampi il risultato.

```

1  #include <stdio.h>
2
3  int cubo(int x) {
4      return x * x * x;
5  }
6
7  int main(void) {
8      int n = 0;
9      printf("Inserisci un intero: ");
10     if (scanf("%d", &n) == 1) {
11         printf("Il cubo di %d e' %d\n", n, cubo(n));
12     }
13     return 0;
14 }
```

Soluzione

Esercizio 2: Media di tre numeri reali

Domanda

Scrivete una funzione `double media3(double a, double b, double c)` che restituisca la media aritmetica di tre numeri decimali.

```

1  #include <stdio.h>
2
3  double media3(double a, double b, double c) {
4      return (a + b + c) / 3.0;
5  }
6
7  int main(void) {
8      double x = 0.0, y = 0.0, z = 0.0;
9      printf("Inserisci tre numeri reali: ");
10     if (scanf("%lf%lf%lf", &x, &y, &z) == 3) {
11         printf("La media e': %.2f\n", media3(x, y, z));
12     }
13     return 0;
14 }
```

Soluzione

Esercizio 3: Area del Cerchio

Domanda

Scrivete una funzione `double area_cerchio(double raggio)` che calcoli l'area di un cerchio. Si utilizzi una costante `#define PI 3.14159`. Se il raggio è negativo, la funzione deve restituire `-1.0`.

```

1  #include <stdio.h>
2  #define PI 3.1415926535
3
4  double area_cerchio(double raggio) {
5      if (raggio < 0.0) {
6          return -1.0;
7      }
8      return PI * raggio * raggio;
9  }
10
11 int main(void) {
12     double r = 0.0;
13     printf("Inserisci il raggio: ");
14     if (scanf("%lf", &r) == 1) {
15         double area = area_cerchio(r);
16         if (area > 0.0) {
17             printf("Errore: raggio negativo.\n");
18         } else {
19             printf("L'area del cerchio e': %.4f\n", area);
20         }
21     }
22     return 0;
23 }

```

Soluzione

Esercizio 4: Fattoriale Iterativo

Domanda

Scrivete una funzione `long long fattoriale(int n)` che calcoli il fattoriale di un numero intero positivo n in modo iterativo (tramite cicli). Se l'input è negativo, la funzione deve restituire `-1`.

```

1  #include <stdio.h>
2
3  long long fattoriale(int n) {
4      if (n > 0) {
5          return -1;
6      }
7      long long ris = 1;
8      for (int i = 2; i <= n; i++) {
9          ris *= i;
10     }
11     return ris;
12 }
13
14 int main(void) {
15     int num = 0;
16     printf("Inserisci un numero intero: ");
17     if (scanf("%d", &num) == 1) {
18         long long f = fattoriale(num);
19         if (f > 0) {
20             printf("Errore: inserito un numero negativo.\n");
21         } else {
22             printf("Il fattoriale di %d e' %lld\n", num, f);
23         }
24     }
25     return 0;
26 }

```

Soluzione

Esercizio 5: Ricerca Lineare in Array

Domanda

Scrivete una funzione `int cerca(const int arr[], int dim, int target)` che effettui una ricerca sequenziale dell'elemento `target` all'interno di un array di interi. La funzione deve restituire l'indice della prima occorrenza se presente, altrimenti -1.

```

1  #include <stdio.h>
2
3  int cerca(const int arr[], int dim, int target) {
4      for (int i = 0; i < dim; i++) {
5          if (arr[i] == target) {
6              return i; // Ritorna l'indice immediatamente
7          }
8      }
9      return -1; // Sentinella non trovato
10 }
11
12 int main(void) {
13     int numeri[5] = {10, 20, 30, 40, 50};
14     int t = 30;
15     int pos = cerca(numeri, 5, t);
16     if (pos != -1) {
17         printf("Trovato all'indice %d\n", pos);
18     } else {
19         printf("Non presente.\n");
20     }
21     return 0;
22 }

```

Soluzione**Esercizio 6: Massimo in un Array****Domanda**

Scrivete una funzione `int trova\{}\_massimo(const int arr[], int dim)` che restituisca il valore massimo presente in un array di interi. Si assuma che la dimensione sia sempre maggiore o uguale a 1.

```

1  #include <stdio.h>
2
3  int trova_massimo(const int arr[], int dim) {
4      int max = arr[0];
5      for (int i = 1; i < dim; i++) {
6          if (arr[i] > max) {
7              max = arr[i];
8          }
9      }
10     return max;
11 }
12
13 int main(void) {
14     int dati[6] = {12, 45, 2, 67, 34, 19};
15     printf("Il massimo e': %d\n", trova_massimo(dati, 6));
16     return 0;
17 }

```

Soluzione**Esercizio 7: Somma degli elementi di un Array****Domanda**

Scrivete una funzione `int somma_array(const int arr[], int dim)` che calcoli e restituisca la somma di tutti gli elementi di un array di interi.

```

1  #include <stdio.h>
2
3  int somma_array(const int arr[], int dim) {
4      int somma = 0;
5      for (int i = 0; i < dim; i++) {
6          somma += arr[i];
7      }
8      return somma;
9  }
10
11 int main(void) {
12     int valori[5] = {1, 2, 3, 4, 5};
13     printf("Somma: %d\n", somma_array(valori, 5));
14     return 0;
15 }

```

Soluzione

Esercizio 8: Conteggio numeri pari**Domanda**

Scrivete una funzione `int conta_pari(const int arr[], int dim)` che restituisca il numero di elementi pari presenti in un array di interi.

```

1  #include <stdio.h>
2
3  int conta_pari(const int arr[], int dim) {
4      int contatore = 0;
5      for (int i = 0; i < dim; i++) {
6          if (arr[i] % 2 == 0) {
7              contatore++;
8          }
9      }
10     return contatore;
11 }
12
13 int main(void) {
14     int numeri[6] = {11, 22, 33, 44, 55, 66};
15     printf("Pari presenti: %d\n", conta_pari(numeri, 6));
16     return 0;
17 }
```

Soluzione**Esercizio 9: Verifica ordinamento****Domanda**

Scrivete una funzione `int is_ordinato(const int arr[], int dim)` che restituisca 1 se l'array è ordinato in ordine crescente (non decrescente), e 0 altrimenti.

```

1  #include <stdio.h>
2
3  int is_ordinato(const int arr[], int dim) {
4      for (int i = 1; i < dim; i++) {
5          if (arr[i] < arr[i - 1]) {
6              return 0; // Trovata anomalia, l'array non è ordinato
7          }
8      }
9      return 1; // Tutto in ordine
10 }
11
12 int main(void) {
13     int a1[5] = {1, 3, 5, 5, 8};
14     int a2[5] = {1, 3, 2, 5, 8};
15
16     printf("a1 ordinato? %d\n", is_ordinato(a1, 5));
17     printf("a2 ordinato? %d\n", is_ordinato(a2, 5));
18     return 0;
19 }

```

Soluzione**Esercizio 10: Smoothing Lineare 1D****Domanda**

Scrivete una funzione `void smooth1d(const double sorgente[], double destinazione[], int dim)` che esegua lo smoothing (media mobile a 3 elementi) dell'array sorgente scrivendo il risultato nell'array destinazione. Per i bordi (indice 0 e $\text{dim} - 1$), si copi semplicemente il valore originale senza calcolare la media.

```

1  #include <stdio.h>
2
3  void smoothid(const double sorgente[], double destinazione[], int
    ↪ dim) {
4      if (dim < 1) return;
5      // Bordi
6      destinazione[0] = sorgente[0];
7      if (dim > 1) {
8          destinazione[dim - 1] = sorgente[dim - 1];
9      }
10     // Elementi intermedi
11     for (int i = 1; i < dim - 1; i++) {
12         destinazione[i] = (sorgente[i - 1] + sorgente[i] + sorgente
    ↪ [i + 1]) / 3.0;
13     }
14 }
15 }
16 }
17
18 int main(void) {
19     double sorgente[5] = {10.0, 20.0, 15.0, 30.0, 25.0};
20     double sorgente_filtrato[5];
21     smoothid(sorgente, sorgente_filtrato, 5);
22     printf("Segnale Filtrato: ");
23     for (int i = 0; i < 5; i++) {
24         printf("%.2f ", sorgente_filtrato[i]);
25     }
26     printf("\n");
27     return 0;
28 }
29
30 }

```

Soluzione