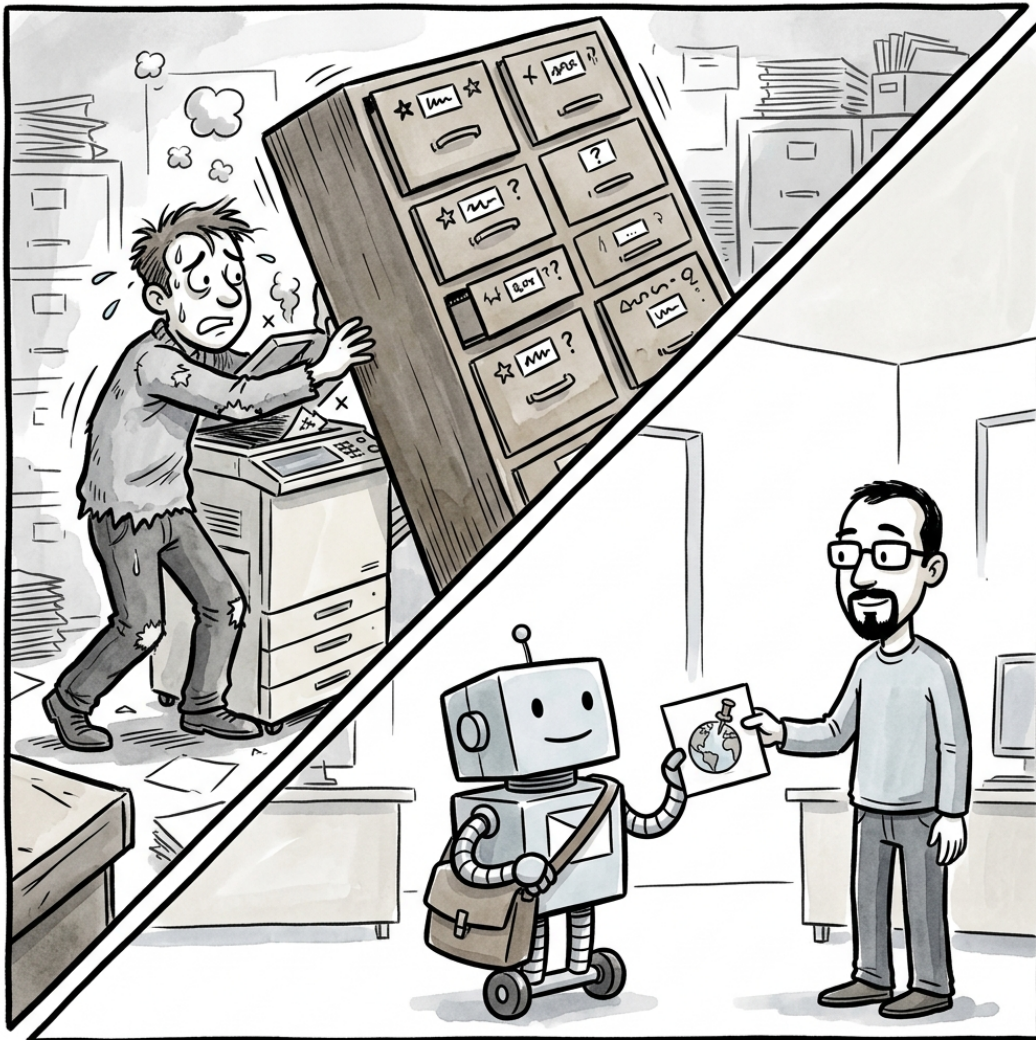


LEZIONE 8

I Puntatori e la Gestione Esplicita della Memoria



Dispense di Programmazione 1 – Canale O-Z

Prof. Antonino Furnari

Corso di Laurea in Informatica

Università di Catania

Lezione 1

I Puntatori e la Gestione Esplicita della Memoria

Nelle lezioni precedenti abbiamo visto come le funzioni siano isolate e ricevano solo “foto-copie” dei dati passati dal chiamante (passaggio per valore). Sebbene questo garantisca sicurezza e pulizia del codice, introduce dei limiti evidenti quando vogliamo che una funzione modifichi direttamente una variabile del chiamante o quando dobbiamo evitare di copiare grandi quantità di dati in memoria. In questa lezione supereremo questo limite entrando nei dettagli fisici della memoria RAM. Introduciamo i **puntatori**, capiremo la reale natura degli array (il decadimento ad array decay), vedremo come eseguire calcoli complessi con l'aritmetica dei puntatori e scopriremo l'ottimizzazione tramite funzioni `inline`.

1.1 La Realtà della RAM e i Tipi Puntatore

1.1.1 Oltre l'illusione delle variabili

Quando scriviamo un programma in C, il compilatore ci offre l'astrazione comoda delle variabili nominate (es. `int x`). Tuttavia, a livello hardware, la CPU non ha idea di cosa sia “x”. La memoria RAM è vista dal processore come una sequenza lineare di celle microscopiche (byte), ciascuna identificata da un **indirizzo fisico** numerico univoco (solitamente espresso in notazione esadecimale, es. `0x7fffd9b3`).

Durante l'esecuzione, lo Stack Pointer alloca i byte necessari nello stack per memorizzare le nostre variabili locali. Se conosciamo l'indirizzo esatto di una di queste celle, possiamo scavalcare l'isolamento dei record di attivazione e scrivervi o leggervi direttamente, a prescindere dal fatto che la variabile appartenga a un'altra funzione.

Il tipo di dato che ci permette di memorizzare e manipolare questi indirizzi di memoria è il **puntatore**.

Per comprendere meglio questo concetto, osserviamo cosa accade durante la traduzione del codice da C ad Assembly (x86-64).

Quello che scriviamo in C:

```
1 int x = 42;  
2 x = 100;
```

Quello che vede la CPU (Assembly x86-64):

```
1 ; La variabile "x" a livello macchina non esiste più. Esiste solo una
2 ; coordinata (offset) calcolata a partire dalla base dello stack frame (
   ⇨ rbp).
3 mov DWORD PTR [rbp-4], 42 ; Scrivi il valore 42 nella cella a indirizzo
   ⇨ (RBP - 4)
4 mov DWORD PTR [rbp-4], 100 ; Sovrascrivi con 100 nella stessa identica
   ⇨ cella
```

Come si può notare, l'identificatore `x` è stato rimpiazzato dall'indirizzo relativo `[rbp-4]`.

1.1.2 Dichiarazione e interpretazione dei puntatori

Un puntatore è a tutti gli effetti una variabile che contiene un indirizzo di memoria. In un'architettura moderna a 64 bit, gli indirizzi sono lunghi 64 bit, quindi qualsiasi variabile puntatore occupa esattamente **8 byte** in memoria.

La sintassi di dichiarazione prevede l'uso dell'asterisco `*`:

```
1 int *p; // p ? un puntatore ad un intero
2 double *q; // q ? un puntatore a un double
```

Si consiglia di leggere questa dichiarazione da destra a sinistra: "*p è un puntatore (*) a un intero (int)*".

Perché i puntatori sono tipizzati?

Se tutti gli indirizzi occupano 8 byte, perché dobbiamo dichiarare `int *p` invece di un ipotetico tipo generico `pointer p`?

La risposta risiede nell'**interpretazione** dei dati: il tipo indica al compilatore come deve essere letta e modificata la memoria una volta giunti a quell'indirizzo.

- Con `char *p`, il compilatore leggerà o scriverà 1 byte a partire dall'indirizzo memorizzato.
- Con `int *p`, leggerà o scriverà 4 byte consecutivi.
- Con `double *p`, interpreterà 8 byte consecutivi secondo lo standard IEEE 754.

Per visualizzare questo comportamento in memoria, consideriamo il seguente codice:

```
1 // Variabili Standard
2 int x = 5;
3 double y = 8.2;
4 char z = 'a';
5
6 // Variabili Puntatore
7 int *px = &x;
8 double *py = &y;
9 char *pz = &z;
```

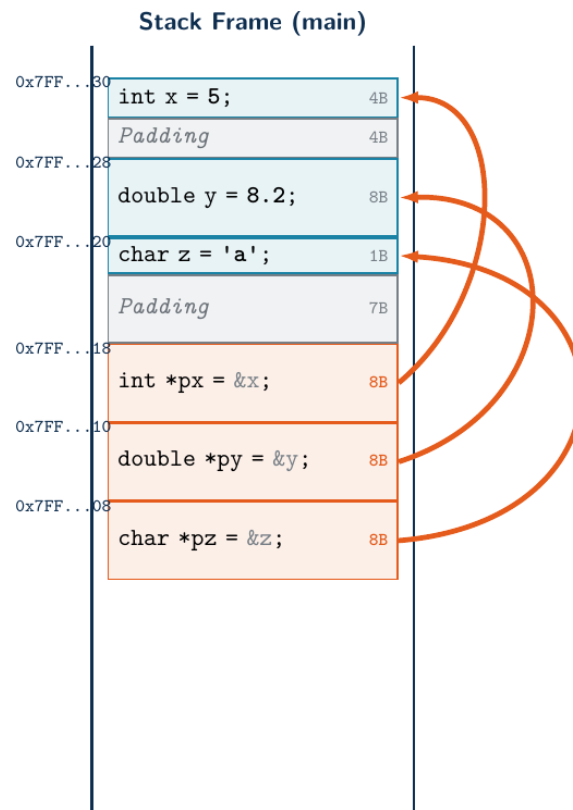


Figura 1.1: Diagramma: L'Anatomia dei Puntatori nello Stack

A livello di Stack Frame, le variabili occupano spazi differenti a seconda del tipo (1, 4 o 8 byte), mentre tutti i puntatori (`px`, `py`, `pz`) occupano esattamente **8 byte**, indipendentemente dal tipo di dato a cui puntano:

Trappola sintattica

Prestare attenzione a dichiarazioni multiple sulla stessa riga:

```
1 int* p, q; // Crea p come puntatore a int, ma q come normale intero!
```

L'asterisco si lega solo all'identificatore immediatamente successivo. La best practice raccomanda di dichiarare un solo puntatore per riga:

```
1 int *p = NULL;  
2 int q = 0;
```

Puntatori a vuoto (NULL)

Dichiarare un puntatore senza inizializzarlo lo rende un *dangling pointer* (puntatore selvaggio), contenente un indirizzo spazzatura casuale. Tentarne l'accesso provocherà un crash improvviso (Segmentation Fault).

È regola aurea inizializzare sempre i puntatori non associati a NULL (o 0):

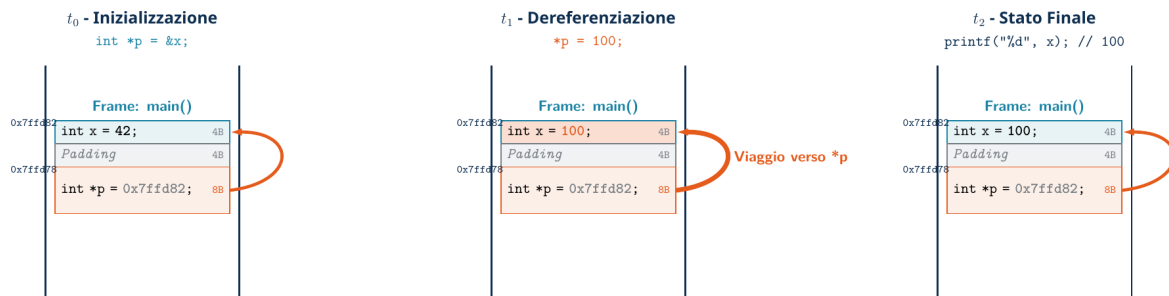


Figura 1.2: Diagramma: L'Operatore di Dereferenziazione

Inizializzare un puntatore a NULL (che sotto il cofano corrisponde all'indirizzo 0) è una fondamentale pratica di programmazione difensiva. L'indirizzo 0 è riservato dal sistema operativo e non è accessibile per la lettura o la scrittura diretta da parte del programma utente.

Impostando un puntatore a NULL dichiariamo esplicitamente: *"questo puntatore al momento è vuoto, non punta a nulla di valido"*. Questo ci permette di implementare controlli logici nel codice per verificare se il puntatore contiene o meno un indirizzo utilizzabile prima di effettuare accessi (evitando di andare incontro a crash del programma).

```
1 int *ptr = NULL; // ptr non punta a nessun indirizzo valido
```

1.1.3 Operatori & (Indirizzo) e * (Dereferenziazione)

Per lavorare con i puntatori utilizziamo due operatori unari fondamentali:

1. **Operatore Indirizzo (&):** Restituisce la coordinata di memoria (l'indirizzo) di una variabile esistente.
2. **Operatore di Dereferenziazione (*):** Consente di accedere al valore memorizzato all'indirizzo puntato, sia in lettura che in scrittura.

```
1 #include <stdio.h>
2
3 int main(void) {
4     int x = 42;
5     int *p = &x; // p memorizza l'indirizzo di x
6
7     printf("Valore di x: %d\n", x);           // Stampa 42
8     printf("Indirizzo di x: %p\n", (void*)p); // Stampa es. 0x7ffd82
9
10    *p = 100; // Dereferenziazione: vai all'indirizzo in p e scrivi 100
11
12    printf("Nuovo valore di x: %d\n", x);      // Stampa 100!
13    return 0;
14 }
```

A livello di Stack Frame, l'istruzione di dereferenziazione si traduce in un accesso diretto alle celle originali:

Sicurezza nell'accesso

Prima di dereferenziare un puntatore, possiamo sfruttare l'inizializzazione a NULL per assicurarci che l'indirizzo sia valido:

```
1 if (ptr != NULL) {
2     *ptr = 10; // Sicuro
3 }
```

Esempio di Crash: Dimenticare il controllo di sicurezza

Cosa succede se omettiamo questo controllo in un programma più strutturato? Consideriamo questo esempio non banale in cui una funzione elabora i dati di un sensore passati tramite puntatore:

```
1 #include <stdio.h>
2
3 void elabora_sensore(const int *dati, int *risultato) {
4     // BUG: Mancano i controlli di sicurezza!
5     // Se dati o risultato sono NULL, avremo un crash.
6     *risultato = (*dati) * 2;
7 }
8
9 int main(void) {
10     int sensor_val = 25;
11     int output = 0;
12
13     // Caso d'uso corretto:
14     elabora_sensore(&sensor_val, &output);
15     printf("Risultato 1: %d\n", output);
16
17     // Caso d'uso errato (ad esempio, sensore scollegato o guasto):
18     int *sensore_guasto = NULL;
19     elabora_sensore(sensore_guasto, &output); // Qui il programma andr?
20     ↪ in crash!
21
22     printf("Risultato 2: %d\n", output);
23     return 0;
24 }
```

Compilando ed eseguendo questo programma da terminale, l'esecuzione si interromperà bruscamente restituendo un errore del tipo `Segmentation fault (core dumped)` (o Errore di segmentazione), senza dare alcuna indicazione su quale riga di codice abbia causato il problema.

Come individuare il crash con il Debugger (LLDB)

Per trovare la riga esatta del crash, possiamo utilizzare il debugger **LLDB** integrato. Ecco i passi da seguire da terminale:

1. Compilare con i simboli di debug:

Dobbiamo indicare al compilatore `gcc` (o `clang`) di includere i metadati di debug con l'opzione `-g`:

```
1 gcc -g sensore.c -o sensore
```

2. Avviare LLDB:

Lanciamo il debugger sul file eseguibile appena generato:

```
1 lldb ./sensore
```

3. Eseguire il programma:

All'interno dell'interfaccia interattiva di LLDB, digitiamo run (o semplicemente r) per avviare l'esecuzione:

```
1 (lldb) run
```

4. Analizzare il punto di arresto:

Il programma verrà eseguito finché non incontrerà l'errore di memoria. LLDB intercetterà il crash e ci mostrerà l'esatta riga di codice che lo ha causato:

```
1 Process 12345 stopped
2
3 * thread #1, queue = 'com.apple.main-thread', stop reason =
  ↳ EXC_BAD_ACCESS (code=1, address=0x0)
4 frame #0: 0x0000000100000f58 sensore`elabora_sensore(dati=0
  ↳ x0000000000000000, risultato=0x00007ff7bfefef08) at sensore.c
  ↳ :6:18
5     3 void elabora_sensore(const int *dati, int *risultato) {
6     4         // BUG: Mancano i controlli di sicurezza!
7     5         // Se dati o risultato sono NULL, avremo un crash.
8     -> 6         *risultato = (*dati) * 2;
9     7     }
```

L'output di LLDB ci dice che:

- Lo stop è avvenuto alla riga 6 del file sensore.c all'interno della funzione elabora\{\}_sensore.
- Il motivo dello stop è EXC\{\}_BAD\{\}_ACCESS (un tentativo di accedere a una zona di memoria non consentita).
- Il parametro dati è impostato a 0x0000000000000000 (ovvero NULL).

5. Esaminare lo Stack dei Frame:

Digitando il comando backtrace (o bt), possiamo vedere l'intera catena di chiamate che ha portato a quella riga, capendo che la funzione elabora\{\}_sensore è stata invocata dal main alla riga 19:

```
1 (lldb) backtrace
2
3 * frame #0: 0x0000000100000f58 sensore`elabora_sensore(dati=0
  ↳ x0000000000000000, risultato=0x00007ff7bfefef08) at sensore.c
  ↳ :6:18
4 frame #1: 0x0000000100000fc4 sensore`main at sensore.c:19:5
```

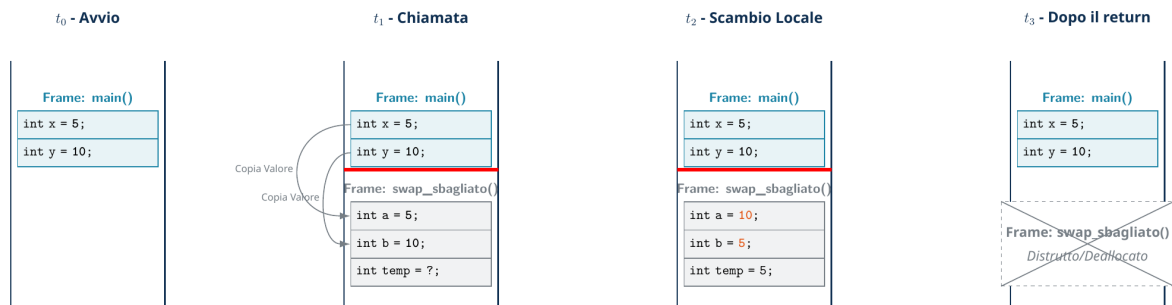


Figura 1.3: Diagramma: Swap Pass-by-Value

6. Risolvere il Bug:

Una volta individuata l'origine, possiamo correggere la funzione aggiungendo la guardia `if`:

```
1 void elabora_sensore(const int *dati, int *risultato) {
2     if (dati == NULL || risultato == NULL) {
3         printf("Errore: puntatori non validi.\n");
4         return;
5     }
6     *risultato = (*dati) * 2;
7 }
```

Infine usciamo dal debugger digitando `quit` o `q`.

1.2 Passaggio per Riferimento e la funzione Swap

1.2.1 Il limite del passaggio per valore

Proviamo a scrivere un programma che inverte due variabili intere usando una funzione `swap`:

```
1 #include <stdio.h>
2
3 void swap_sbagliato(int a, int b) {
4     int temp = a;
5     a = b;
6     b = temp;
7 }
8
9 int main(void) {
10     int x = 5;
11     int y = 10;
12     swap_sbagliato(x, y); // Passa i valori di x e y (fotocopie)
13     printf("x = %d, y = %d\n", x, y); // Stampa x = 5, y = 10!
14     return 0;
15 }
```

Se eseguiamo questo codice, il valore di `x` e `y` nel `main` rimarrà rispettivamente 5 e 10. Questo perché la funzione lavora su copie isolate collocate nel proprio Stack Frame, che vengono distrutte non appena la funzione termina.

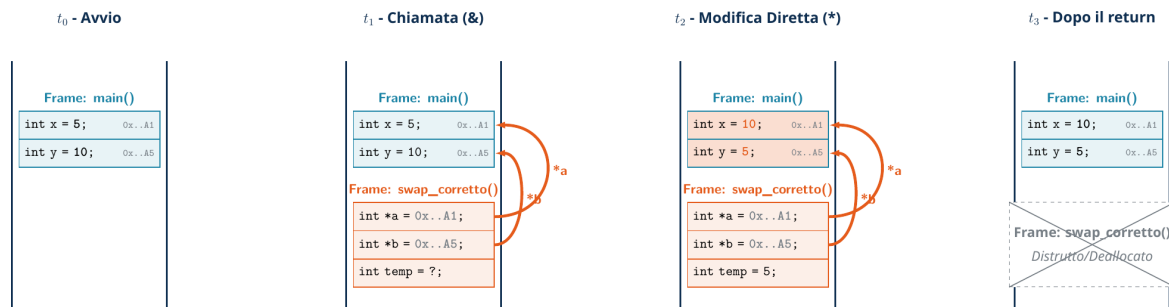


Figura 1.4: Diagramma: Swap Pass-by-Reference

1.2.2 Il passaggio per riferimento simulato

Per consentire a una funzione di modificare le variabili del chiamante, dobbiamo passarle gli **indirizzi** delle variabili (riferimento simulato tramite puntatori):

```

1  #include <stdio.h>
2
3  void swap_corretto(int *a, int *b) {
4      int temp = *a; // Copia il valore puntato da a
5      *a = *b;       // Scrive il valore puntato da b all'indirizzo a
6      *b = temp;     // Scrive temp all'indirizzo b
7  }
8
9  int main(void) {
10     int x = 5, y = 10;
11     swap_corretto(&x, &y); // Passa gli indirizzi di x e y
12     printf("x = %d, y = %d\n", x, y); // Stampa x = 10, y = 5!
13     return 0;
14 }

```

Grazie ai puntatori, swap\{\}_corretto può raggiungere direttamente la memoria riservata al main e modificare fisicamente le variabili originali.

1.2.3 La corrispondenza tra Array e Puntatori

In C esiste un legame indissolubile tra gli array e i puntatori. Quando dichiariamo un array statico:

```

1  int arr[5] = {10, 20, 30, 40, 50};

```

Il nome dell'array, `arr`, si comporta a tutti gli effetti come un **puntatore costante al suo primo elemento**. In altre parole, l'espressione `arr` è semanticamente equivalente a `&arr[0]`.

Questa corrispondenza si riflette anche nella sintassi di accesso. Quando scriviamo `arr[i]`, il compilatore traduce questa notazione in un'operazione di aritmetica dei puntatori seguita da dereferenziazione:

```

1  arr[i]  <==>  *(arr + i)

```

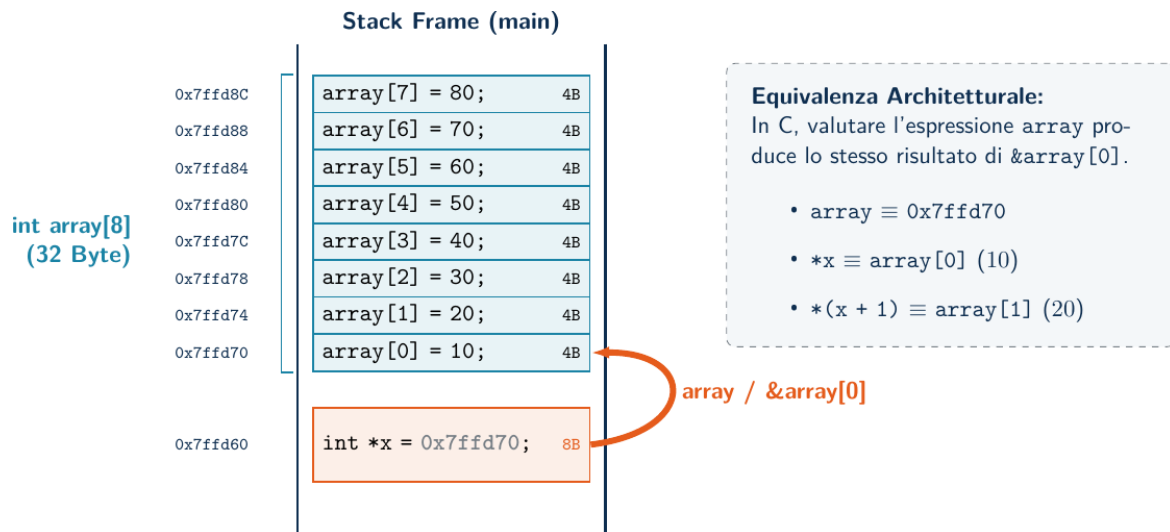


Figura 1.5: Diagramma: Equivalenza Array e Puntatori

Ciò significa che possiamo scorrere un array sia utilizzando l'indice `[i]` sia manipolando direttamente l'indirizzo tramite l'aritmetica dei puntatori.

Per chiarire questo legame intimo tra array e puntatori, osserviamo il seguente codice d'esempio:

```

1  #include <stdio.h>
2
3  int main(void) {
4      // Allocazione contigua di 8 interi (32 Byte)
5      int array[8] = {10, 20, 30, 40, 50, 60, 70, 80};
6
7      // x memorizza l'indirizzo di partenza dell'array
8      int *x = array;
9
10     // Equivalenza fondamentale:
11     // array == &array[0]
12     printf("Indirizzo array: %p\n", (void*)array);
13     printf("Indirizzo primo elemento: %p\n", (void*)&array[0]);
14     printf("Valore puntato da x: %d\n", *x); // Stampa 10
15
16     return 0;
17 }
```

A livello dello Stack Frame, le celle dell'array sono allocate in posizioni consecutive di memoria, e il puntatore `x` punta all'inizio di questa sequenza:

1.2.4 Il decadimento degli array (Array Decay)

Quando passiamo un array come parametro a una funzione, il linguaggio C non copia l'intero array in memoria (operazione che richiederebbe molto tempo e memoria per array di grandi dimensioni). Invece, applica la regola del **decadimento degli array (array decay)**: l'array decade a un semplice puntatore al suo primo elemento.

Di conseguenza, qualsiasi parametro formale dichiarato come `int arr[]` o `int arr[100]` viene interpretato dal compilatore come `int *arr`.

Questo ha un effetto collaterale importante: **la funzione perde l'informazione sulla dimensione dell'array**. Possiamo verificarlo empiricamente confrontando l'output dell'operatore `sizeof` all'interno del `main` e all'interno di una funzione:

```
1 #include <stdio.h>
2
3 void stampa_dimensione(int arr[], int dim) {
4     // All'interno della funzione, arr ? diventato un semplice puntatore
5     // sizeof(arr) restituisce la dimensione di un puntatore (8 byte su
6     // sistemi a 64-bit).
7     printf("In funzione: sizeof(arr) = %lu byte\n", sizeof(arr));
8 }
9
10 int main(void) {
11     int voti[5] = {18, 22, 25, 28, 30};
12
13     // Nel main, voti ? un array statico di 5 interi.
14     // sizeof(voti) restituisce la dimensione totale: 5 * sizeof(int) =
15     // 20 byte.
16     printf("Nel main: sizeof(voti) = %lu byte\n", sizeof(voti));
17
18     stampa_dimensione(voti, 5);
19     return 0;
20 }
```

Output del programma (su sistema a 64-bit):

```
1 Nel main: sizeof(voti) = 20 byte
2 In funzione: sizeof(arr) = 8 byte
```

Per questo motivo, in C è **sempre necessario** passare la dimensione dell'array come parametro aggiuntivo (`int dim`) a qualsiasi funzione debba elaborarlo.

1.2.5 Aritmetica dei puntatori e zucchero sintattico

Sui puntatori è possibile eseguire operazioni aritmetiche (come la somma e la sottrazione). L'aritmetica dei puntatori è **tipizzata**: aggiungere 1 a un puntatore non aumenta l'indirizzo di 1 byte, ma di una quantità pari a `sizeof(tipo)` associato al puntatore (ovvero la dimensione del tipo a cui punta).

Consideriamo il confronto nel codice seguente:

Aritmetica su Interi (`sizeof(int) == 4`):

```
1 int i[4] = {10, 20, 30, 40};
2 int *p = i; // p punta a i[0] (es. indirizzo 1000)
3
4 // p + 1 incrementa l'indirizzo di 1 * sizeof(int) = 4 byte
5 // p + 2 incrementa l'indirizzo di 2 * sizeof(int) = 8 byte
6 printf("p : %p\n", (void*)p); // 1000
7 printf("p+1 : %p\n", (void*)(p + 1)); // 1004
```

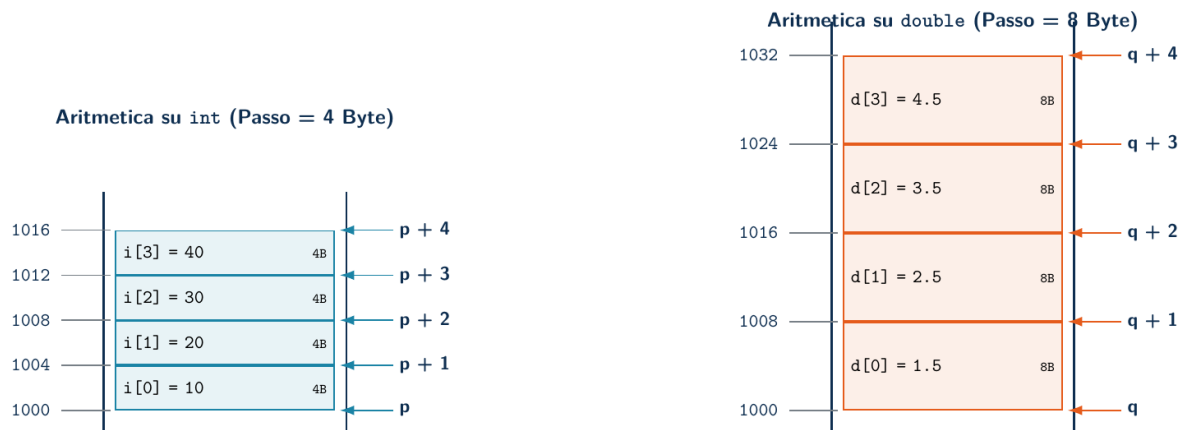


Figura 1.6: Diagramma: Aritmetica dei Puntatori

Aritmetica su Reali (sizeof(double) == 8):

```

1 double d[4] = {1.5, 2.5, 3.5, 4.5};
2 double *q = d; // q punta a d[0] (es. indirizzo 1000)
3
4 // q + 1 incrementa l'indirizzo di 1 * sizeof(double) = 8 byte
5 // q + 2 incrementa l'indirizzo di 2 * sizeof(double) = 16 byte
6 printf("q : %p\n", (void*)q); // 1000
7 printf("q+1 : %p\n", (void*)(q + 1)); // 1008

```

A livello fisico, la CPU calcola l'indirizzo sommando al valore del puntatore l'indice moltiplicato per la dimensione del tipo di dato. Il passo di incremento (passo dell'aritmetica) si adatta quindi automaticamente al tipo di puntatore:

Questo spiega perché in C l'accesso agli elementi sia estremamente veloce: si tratta di una semplice operazione aritmetica di offset seguita da dereferenziazione.

1.2.6 Sicurezza: Il modificatore const**Perché non possiamo cambiare l'indirizzo di un array**

Come abbiamo detto, il nome dell'array `arr` è un puntatore al primo elemento, ma con una restrizione cruciale: **è un puntatore costante**. Questo significa che il compilatore fissa l'indirizzo base dell'array al momento dell'allocazione e non permette di cambiarlo. Un'istruzione come la seguente genererà un errore di compilazione:

```

1 int x = 42;
2 int arr[5];
3 arr = &x; // ERRORE DI COMPILAZIONE! arr equivale a un int * const, non
            // pu? essere riassegnato

```

Proteggere i dati in lettura con const

Poiché passare un array (ovvero un puntatore al suo primo elemento) a una funzione dà a quest'ultima il potere di modificarne il contenuto originale, dobbiamo poter proteggere i dati quando vogliamo che la funzione si limiti a leggerli.

Per farlo, dichiariamo il parametro formale come puntatore a costante utilizzando la parola chiave `const`:

```
1 void stampa_array(const int *arr, int dim) {  
2     // arr[0] = 99; // ERRORE DI COMPILAZIONE! Il compilatore impedisce  
   ↪ la scrittura  
3 }
```

- `const int *p`: Il puntatore può cambiare indirizzo, ma il dato puntato è costante (sola lettura).
- `int * const p`: Il dato puntato può essere modificato, ma il puntatore non può cambiare indirizzo (comportamento nativo degli array).
- `const int * const p`: Sia il puntatore che il dato puntato sono costanti.

1.2.7 Funzioni inline

Le chiamate a funzione richiedono un costo architetturale (overhead) per la creazione e distruzione dei record di attivazione nello stack. Per funzioni piccolissime invocate frequentemente (es. all'interno di cicli intensivi), questo overhead può impattare sulle prestazioni.

La keyword `inline` suggerisce al compilatore di sostituire la chiamata con il codice effettivo direttamente nel punto in cui è invocata:

```
1 static inline int quadrato(int x) {  
2     return x * x;  
3 }
```

Questo elimina totalmente l'overhead della chiamata a funzione senza perdere i vantaggi di modularità e leggibilità del codice.

Per comprendere concretamente l'effetto dell'inlining, osserviamo a confronto il codice C e l'Assembly generato da una funzione `static inline` (in cui il compilatore ha scelto di non espandere a causa di mancate ottimizzazioni) rispetto ad una funzione classica, notando la chiamata effettiva tramite `call`.

Caso 1: Utilizzo di `static inline` (senza ottimizzazioni attive)

Se scriviamo una funzione dichiarata come `static inline`, in assenza di flag di ottimizzazione del compilatore, quest'ultimo potrebbe comunque generare una chiamata a funzione standard.

Attenzione

Perché `static inline` e non solo `inline`? In C99 una funzione dichiarata solo `inline` è una "definizione inline" che **non** produce il codice della funzione nel file oggetto: se il compilatore (ad esempio senza `-O2`) decide di non espandere la chiamata, il linker non trova la funzione e fallisce con `undefined reference to 'add'`. Aggiungendo `static` la funzione diventa locale al file e il compilatore ne genera sempre una copia

quando serve.

Codice C:

```

1  #include <stdio.h>
2
3  static inline int add(int x, int y) {
4      return x + y;
5  }
6
7  int main(void) {
8      int x = 3 + add(5, 5);
9      return x;
10 }
```

Assembly x86-64 generato per il main:

```

1  main:
2      push    rbp
3      mov     rbp, rsp
4      sub     rsp, 16
5      mov     dword ptr [rbp - 4], 0
6      mov     esi, 5
7      mov     edi, esi
8      call    add                ; Esegue comunque la chiamata alla
   ↪ funzione (locale) add
9      add     eax, 3              ; Somma 3 al risultato restituito in eax
10     mov     dword ptr [rbp - 8], eax
11     mov     eax, dword ptr [rbp - 8]
12     add     rsp, 16
13     pop     rbp
14     ret
```

Caso 2: Funzione Classica (Senza inline)

Nel caso di una funzione standard, il compilatore genera separatamente il codice della funzione add e lo invoca esplicitamente nel main.

Codice C:

```

1  #include <stdio.h>
2
3  int add(int x, int y) {
4      return x + y;
5  }
6
7  int main(void) {
8      int x = 3 + add(5, 5);
9      return x;
10 }
```

Assembly x86-64 generato:

```

1  add:
2      push    rbp
3      mov     rbp, rsp
4      mov     dword ptr [rbp - 4], edi ; Primo parametro
5      mov     dword ptr [rbp - 8], esi ; Secondo parametro
6      mov     eax, dword ptr [rbp - 4]
7      add     eax, dword ptr [rbp - 8] ; Esegue la somma
8      pop     rbp
9      ret
10
11 main:
12     push    rbp
13     mov     rbp, rsp
14     sub     rsp, 16
15     mov     dword ptr [rbp - 4], 0
16     mov     esi, 5
17     mov     edi, esi
18     call    add ; Chiama direttamente la
    ↪ funzione add
19     add     eax, 3 ; Somma 3
20     mov     dword ptr [rbp - 8], eax
21     mov     eax, dword ptr [rbp - 8]
22     add     rsp, 16
23     pop     rbp
24     ret

```

Quando invece attiviamo le ottimizzazioni del compilatore (es. `-O2`), la keyword `inline` fa sì che il codice di `add` venga iniettato direttamente all'interno di `main`, traducendo l'intera espressione in un unico calcolo immediato (con `-O2` il `main` diventa semplicemente `mov eax, 13` seguito da `ret`) ed eliminando sia l'istruzione `call` sia lo stack frame associato alla funzione chiamata.

1.3 Quiz di Auto-valutazione

Quiz 1

Domanda

1. Qual è la dimensione in byte di una variabile puntatore su un calcolatore a 64 bit?

- A) Dipende dal tipo a cui punta (es. 4 byte per int, 1 byte per char)
- B) Sempre 8 byte
- C) Sempre 4 byte

Soluzione
Risposta corretta: B
Spiegazione: Un puntatore contiene un indirizzo di memoria. Nelle architetture a 64 bit, l'indirizzo di memoria ha sempre una lunghezza fissa di 64 bit (ovvero 8 byte), indipendentemente dal tipo di dato a cui punta.

Quiz 2

Domanda

2. Quale operatore restituisce l'indirizzo di memoria di una variabile esistente?

- A) *
- B) \{\}&
- C) -->

Soluzione
Risposta corretta: B
Spiegazione: L'operatore unario di indirizzo \{\}& (address-of) estrae le coordinate di memoria fisiche in cui è allocata la variabile.

Quiz 3

Domanda

3. Se ptr è un puntatore a intero inizializzato a NULL, cosa succede se proviamo a eseguire l'istruzione *ptr = 10;?

- A) La variabile intera viene correttamente modificata
- B) Il programma viene compilato correttamente ma va in crash (Segmentation Fault) all'esecuzione
- C) Errore di compilazione immediato

Soluzione
Risposta corretta: B
Spiegazione: Compilare la dereferenziazione di un puntatore è sintatticamente valido. All'atto pratico, tentare di accedere all'indirizzo 0x0 (NULL) genera una violazione di accesso di memoria (Segmentation Fault) gestita dal sistema operativo che termina forzatamente il processo.

Quiz 4

Domanda

4. Perché i puntatori richiedono una tipizzazione (es. int* o char*)?

- A) Per definire lo spazio occupato dal puntatore stesso (8 byte vs 4 byte)
- B) Per consentire al compilatore di sapere quanti byte leggere/scrivere e come interpretarli all'indirizzo puntato
- C) È solo un vincolo estetico del linguaggio C

Soluzione
Risposta corretta: B
Spiegazione: Sebbene l'indirizzo occupi sempre 8 byte, il tipo associato specifica la "lente" attraverso cui guardare la cella: definisce quanti byte consecutivi considerare a partire da quell'indirizzo e con quale schema interpretarli.

Quiz 5

Domanda

5. Cosa significa il termine "Array Decay" in C?

- A) La progressiva perdita di precisione dei dati all'interno dell'array
- B) Il fatto che il nome di un array decade implicitamente a un puntatore al suo primo elemento quando passato come parametro
- C) La distruzione automatica dell'array alla fine del ciclo di vita

Soluzione
Risposta corretta: B
Spiegazione: Quando un array viene passato come argomento a una funzione, decade in un semplice puntatore (`\{\}\&arr[0]`), perdendo l'informazione sulla sua dimensione totale originale (`sizeof`).

Quiz 6

Domanda

6. Se un puntatore a intero p memorizza l'indirizzo decimale 2000, quale indirizzo conterrà l'espressione $p + 2$?

- A) 2002
- B) 2008
- C) 2016

Soluzione
Risposta corretta: B
Spiegazione: L'aritmetica dei puntatori scala l'addizione per la dimensione del tipo a cui punta. Poiché in genere un intero (`int`) occupa 4 byte, $p + 2$ equivale a $2000 + (2 * 4) = 2008$.

Quiz 7

Domanda

7. Qual è l'equivalenza fondamentale che lega gli array e la dereferenziazione in C?

- A) `arr[i]` equivale a `*(arr + i)`
- B) `arr[i]` equivale a `{&arr + i}`
- C) `arr[i]` equivale a `*arr + i`

Soluzione
Risposta corretta: A
Spiegazione: La notazione con parentesi quadre è solo zucchero sintattico per l'aritmetica dei puntatori. `arr[i]` esegue una somma sull'indirizzo base `arr` con offset `i` e dereferenzia il risultato.

Quiz 8

Domanda

8. Cosa dichiara l'istruzione `const int *ptr = &x;`?

- A) Un puntatore costante: non possiamo spostare `ptr` su altri indirizzi
- B) Un puntatore a un intero costante: `ptr` può puntare altrove, ma non possiamo modificare il valore di `x` tramite `ptr`
- C) Tutto bloccato: sia `ptr` che `x` sono immutabili

Soluzione
Risposta corretta: B
Spiegazione: La parola chiave `const` prima dell'asterisco protegge il valore puntato. Il puntatore stesso rimane modificabile (`ptr = &y;` è valido), ma il contenuto all'indirizzo puntato diventa in sola lettura.

Quiz 9

Domanda

9. A cosa serve la parola chiave `inline` davanti alla definizione di una funzione?

- A) A dire al compilatore di inserire il corpo della funzione direttamente nel punto di chiamata, riducendo l'overhead dello stack frame
- B) A consentire alla funzione di essere dichiarata su un'unica riga
- C) A forzare la funzione a essere compilata solo per sistemi multi-thread

Soluzione
Risposta corretta: A
Spiegazione: La keyword `inline` suggerisce al compilatore di sostituire la chiamata con il codice vero e proprio, eliminando l'overhead computazionale dovuto a push e pop del record di attivazione dello Stack.

Quiz 10

Domanda

10. Qual è il comportamento atteso dichiarando un parametro formale come `int * const arr`?

- A) L'array è di sola lettura
- B) L'indirizzo puntato da `arr` non può essere modificato (`arr` è costante), ma gli elementi all'interno possono essere cambiati
- C) Il compilatore genera un errore di sintassi

Soluzione: Collocare `const` dopo l'asterisco rende il puntatore stesso costante (sola scrittura dell'indirizzo). Gli elementi a cui punta restano accessibili in scrittura.

Risposta corretta: B

Soluzione

1.4 Esercizi Pratici

Esercizio 1: Incremento tramite Puntatore

Domanda

Scrivete una funzione `void incrementa(int *x)` che incrementi di una unità il valore contenuto all'indirizzo puntato da `x`. Nel `main`, dichiarate un intero, chiamate la funzione passandone l'indirizzo e verificate il cambiamento.

```

1  #include <stdio.h>
2
3  void incrementa(int *x) {
4      if (x != NULL) {
5          (*x)++; // Nota: le parentesi sono necessarie a causa delle
6                  ↪ precedenza
7      }
8  }
9  int main(void) {
10     int valore = 10;
11     printf("Valore originale: %d\n", valore);
12     incrementa(&valore);
13     printf("Valore incrementato: %d\n", valore); // Stampare? 11
14     return 0;
15 }
```

Soluzione

Esercizio 2: Prodotto di due numeri**Domanda**

Scrivete una funzione `double calcola_prodotto(const double *a, const double *b)` che calcoli e restituisca il prodotto dei due valori reali puntati da `a` e `b`. I valori originali non devono essere modificabili all'interno della funzione.

```

1  #include <stdio.h>
2
3  double calcola_prodotto(const double *a, const double *b) {
4      if (a == NULL || b == NULL)
5          return (a) * (b);
6      }
7      return 0.0;
8  }
9
10 int main(void) {
11     double n1 = 3.5;
12     double n2 = 2.0;
13     double ris = calcola_prodotto(&n1, &n2);
14     printf("Prodotto: %.2f\n", ris); // Stampa: 7.00
15     return 0;
16 }
```

Soluzione**Esercizio 3: Swap di due Double****Domanda**

Scrivete una funzione `void swap_double(double *x, double *y)` che scambi i valori reali contenuti in due variabili reali del chiamante.

```

1  #include <stdio.h>
2
3  void swap_double(double *x, double *y) {
4      if (x == NULL || y == NULL) return;
5      double temp = *x;
6      *x = *y;
7      *y = temp;
8  }
9
10 int main(void) {
11     double a = 1.23, b = 9.87;
12     printf("Prima: a=%.2f, b=%.2f\n", a, b);
13     swap_double(&a, &b);
14     printf("Dopo: a=%.2f, b=%.2f\n", a, b);
15     return 0;
16 }

```

Soluzione

Esercizio 4: Azzera Array tramite Puntatore

Domanda

Scrivete una funzione `void azzera_array(int *arr, int dim)` che azzeri tutti gli elementi di un array di interi. Utilizzate l'aritmetica dei puntatori per scorrere l'array invece degli indici quadrate (zucchero sintattico).

```

1  #include <stdio.h>
2
3  void azzera_array(int *arr, int dim) {
4      if (arr == NULL) return;
5      for (int i = 0; i < dim; i++) {
6          *(arr + i) = 0; // Uso dell'aritmetica dei puntatori
7      }
8  }
9
10 int main(void) {
11     int numeri[5] = {1, 2, 3, 4, 5};
12     azzera_array(numeri, 5);
13     for (int i = 0; i < 5; i++) {
14         printf("%d ", numeri[i]); // Stampare: 0 0 0 0 0
15     }
16     printf("\n");
17     return 0;
18 }

```

Soluzione

Esercizio 5: Riferimento a minimo (Doppio Puntatore)**Domanda**

Scrivete una funzione `void trova_min_ptr(const int *arr, int dim, const int **min_ptr)` che trovi l'indirizzo dell'elemento minimo all'interno di un array e scriva tale indirizzo nella variabile puntata da `min_ptr`.

```

1  #include <stdio.h>
2
3  void trova_min_ptr(const int *arr, int dim, const int **min_ptr) {
4      if (arr == NULL || min_ptr == NULL || dim <= 0) return;
5
6      const int *min_corrente = arr; // Inizializza al primo elemento
7      for (int i = 1; i < dim; i++) {
8          if (*(arr + i) < *min_corrente)
9              min_corrente = arr + i;
10     }
11     *min_ptr = min_corrente; // Scrive l'indirizzo trovato
12 }
13
14 int main(void) {
15     int dati[5] = {34, 12, 56, 8, 45};
16     const int *min_addr = NULL;
17
18     trova_min_ptr(dati, 5, &min_addr);
19
20     if (min_addr != NULL)
21         printf("Il minimo e': %d, all'indirizzo %p\n", *min_addr, (
22         void *)min_addr);
23     }
24     return 0;
25 }

```

Soluzione**Esercizio 6: Inversione di un array in-place con due****Domanda**

puntatori

Scrivete una funzione `void inverti_array(int *arr, int dim)` che inverta gli elementi di un array sul posto usando due puntatori di supporto: uno impostato al primo elemento, l'altro all'ultimo elemento. Si effettui lo swap degli elementi e si muovano i puntatori l'uno verso l'altro finché non si incontrano.

```

1  #include <stdio.h>
2
3  void inverti_array(int *arr, int dim)
4  {
5      if (arr == NULL || dim <= 1) return;
6
7      int *sinistra = arr;
8      int *destra = arr + dim - 1;
9
10     while (sinistra < destra)
11     {
12         // Swap degli elementi puntati
13         int temp = *sinistra;
14         *sinistra = *destra;
15         *destra = temp;
16
17         // Spostamento dei puntatori
18         sinistra++;
19         destra--;
20     }
21
22     int v[6] = {10, 20, 30, 40, 50, 60};
23     inverti_array(v, 6);
24     for (int i = 0; i < 6; i++) {
25         printf("%d ", v[i]); // Stampo: 60 50 40 30 20 10
26     }
27     printf("\n");
28     return 0;
29 }

```

Soluzione**Esercizio 7: Confronto di Puntatori****Domanda**

Scrivete una funzione `int confronta({}_puntatori(const int *p1, const int *p2)` che verifichi i valori puntati. Restituisca 1 se il valore puntato da `p1` è maggiore di quello di `p2`, -1 se è minore, e 0 se sono uguali. Se uno dei due puntatori è nullo, la funzione deve restituire -999.

```

1  #include <stdio.h>
2
3  int confronta_puntatori(const int *p1, const int *p2) {
4      if (p1 == NULL || p2 == NULL)
5          return -999;
6      }
7      if (*p2 < *p1)
8          return 1;
9      if (*p2 > *p1)
10         return -1;
11     return 0;
12 }
13
14 int main(void) {
15     int a = 15, b = 20;
16     printf("Risultato: %d\n", confronta_puntatori(&a, &b)); //
17     // Stampare? -1
18     printf("Risultato NULL: %d\n", confronta_puntatori(&a, NULL));
19     // Stampare? -999
20     return 0;
21 }

```

Soluzione**Esercizio 8: Lunghezza stringa con Aritmetica dei****Domanda****Puntatori**

Scrivete una funzione `int calcola_lunghezza_stringa(const char *s)` che calcoli la lunghezza di una stringa in C (terminata dal carattere `\0`) incrementando un puntatore da inizio stringa e calcolando la differenza di indirizzi alla fine. Non utilizzate indici o variabili di conteggio.

```

1  #include <stdio.h>
2
3  int calcola_lunghezza_stringa(const char *s) {
4      if (s == NULL) return 0;
5      const char *p = s;
6      while (*p != '\0') {
7          p++; // Avanza il puntatore
8      }
9      return p - s; // Differenza tra puntatori: restituisce il
10     } // numero di char
11
12     int main(void) {
13         char testo[] = "Puntatori";
14         printf("Lunghezza di '%s': %d\n", testo,
15             ↪ calcola_lunghezza_stringa(testo)); // Stampare? 9
16     }

```

Soluzione**Esercizio 9: Copia Sicura di un Array Constant****Domanda**

Scrivete una funzione `void copia\{\}_array\{\}_const(const int *sorgente, int *destinazione, int dim)` che copi tutti gli elementi da un array sorgente (in sola lettura) a un array di destinazione usando puntatori.

```

1  #include <stdio.h>
2
3  void copia_array_const(const int *sorgente, int *destinazione, int
4      ← dim)
5  {
6      if (sorgente == NULL || destinazione == NULL) return;
7      for (int i = 0; i < dim; i++)
8          *(destinazione + i) = *(sorgente + i);
9  }
10 int main(void) {
11     int src[4] = {5, 10, 15, 20};
12     int dst[4];
13     copia_array_const(src, dst, 4);
14     for (int i = 0; i < 4; i++) {
15         printf("%d ", dst[i]); // Stampare? 5 10 15 20
16     }
17     printf("\n");
18     return 0;
19 }

```

Soluzione**Esercizio 10: Swap di Puntatori****Domanda**

Scrivete una funzione `void swap_{ }_puntatori(int **p1, int **p2)` che scambi gli indirizzi di memoria memorizzati all'interno di due variabili puntatore. Nel `main`, verificate che dopo lo `swap`, i due puntatori puntino a variabili inverse.

Soluzione

```

1  #include <stdio.h>
2
3  void swap_puntatori(int **p1, int **p2) {
4      if (p1 != NULL && p2 != NULL) {
5          int *temp = *p1;
6          *p1 = *p2;
7          *p2 = temp;
8      }
9  }
10
11 int main(void) {
12     int x = 100, y = 200;
13     int *p1 = &x;
14     int *p2 = &y;
15
16     printf("p1 punta a valore: %d, p2 punta a valore: %d\n", *
17         ↪ p1, *p2);
18     swap_puntatori(&p1, &p2); // Scambia gli indirizzi
19         ↪ contenuti in p1 e p2
20     printf("Dopo swap:\n");
21     printf("p1 punta a valore: %d, p2 punta a valore: %d\n", *
22         ↪ p1, *p2);
23     // Stampa? p1 punta a valore 200 e p2 punta a valore 100
24     return 0;
25 }

```