



Programmazione 1 (Canale O-Z)

Lezione 7: L'Astrazione del Codice e l'Anatomia dello Stack

Docente: Prof. Antonino Furnari

Email: `antonino.furnari@unict.it`

Sito Web: antoninofurnari.github.io/prog1-2627

🤔 Tre Array, Tre Massimi

Nella scorsa lezione abbiamo trovato il massimo di un array.

? E se avessimo **tre** array diversi e volessimo il massimo di ciascuno? Come fareste con quello che sappiamo oggi?





Il Problema: Duplicazione del Codice

```
// Ricerca massimo nel primo array
int max1 = array1[0];
for (int i = 1; i < N; i++) {
    if (array1[i] > max1) {
        max1 = array1[i];
    }
}

// Ricerca massimo nel secondo array (COPIA-INCOLLA)
int max2 = array2[0];
for (int i = 1; i < N; i++) {
    if (array2[i] > max2) {
        max2 = array2[i];
    }
}

// E così via per il terzo array...
```

- **Il copia-incolla è il male:** un bug nel ciclo (es. un *off-by-one*) va corretto in **ogni** copia.
- Se ne dimentichiamo una, il programma diventa **inconsistente**.
- **Principio DRY** (*Don't Repeat Yourself*): evitare la duplicazione per migliorare manutenibilità e leggibilità.

La Soluzione: Le Funzioni

Un blocco di codice autonomo, riutilizzabile e dotato di **nome** (paradigma **procedurale**).

- **Le usiamo già:** il `main` stesso è una funzione (il punto d'ingresso). E poi `printf`, `scanf`, `pow`, `sqrt`, `rand`, `srand`.
- **Astrazione:** trattiamo la funzione come una **scatola nera** (*black box*): ci interessa cosa entra e cosa esce, non come è calcolato.
- Le librerie ci danno implementazioni **robuste e testate**; ma se ci serve un calcolo che non esiste in nessuna libreria, scriviamo **funzioni personalizzate** (*user-defined*).

? Cosa dovremmo scrivere per calcolare una radice quadrata **senza** poter usare `sqrt` ?



Senza Funzioni: la Radice Quadrata "a Mano"

```
// Come calcoliamo la radice oggi grazie alle librerie (1 sola riga, zero pensieri):  
double radice = sqrt(25.0);
```

```
// Come dovremmo scriverlo nel main se NON potessimo usare le funzioni:
```

```
double n = 25.0;  
double radice;  
if (n < 0) {  
    printf("Errore: radice di numero negativo!\n");  
    // gestione dell'errore...  
} else {  
    // Algoritmo di approssimazione di Newton-Raphson iterativo  
    double x = n;  
    double y = 1.0;  
    double e = 0.000001;  
    while (x - y > e) {  
        x = (x + y) / 2;  
        y = n / x;  
    }  
    radice = x;  
}
```

```
// E dovremmo fare questo COPIA-INCOLLA ogni volta che ci serve una radice nel programma!
```



Anatomia di una Funzione

Come in matematica, $f : D \rightarrow C$: il **dominio** D sono i tipi dei parametri, il **codominio** C il tipo di ritorno, la legge è il **corpo**.

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

- **Tipo di ritorno:** il tipo dell'output; `void` se la funzione non restituisce nulla.
- **Nome:** stesse regole dei nomi di variabili.
- **Parametri formali:** variabili locali temporanee, popolate al momento della chiamata.
- `return`: termina la funzione e restituisce il valore. Obbligatorio se il tipo non è `void`; nelle `void`, `return`; è facoltativo.



Firma e Definizione: un Esempio

```
// Prototipo o Firma (Il "Cosa")
int calcola_somma(int a, int b);

// Definizione (Il "Come")
int calcola_somma(int a, int b) {
    int risultato = a + b;
    return risultato;
}
```

- La **firma** (tipo di ritorno, nome, parametri) definisce l'**interfaccia**.
- Il **corpo** tra graffe definisce il **comportamento**.

? Il compilatore legge il file dall'alto verso il basso: cosa succede se il `main` chiama una funzione definita più in basso?

⚙️ Prototipi e Definizioni

```
#include <stdio.h>

// Dichiarazione (Prototipo)
double half(int x);

int main(void) {
    int val = 5;
    double ris = half(val); // Chiamata
    printf("La meta' e': %f\n", ris);
    return 0;
}

// Definizione
double half(int x) {
    return (double)x / 2;
}
```

- **Prototipi** in cima al file, **definizioni** in fondo.
- Nel prototipo i nomi dei parametri sono **facoltativi**: `double half(int);` è valido quanto `double half(int x);`.
- In assembly non esistono nomi, solo indirizzi, registri e offset.

🎬 **Live:** `gcc -Wall half.c -o half && ./half` ➡ La meta' e': 2.500000. Poi cancellare il prototipo, ricompilare e leggere insieme il messaggio del compilatore.

Il Flusso di Esecuzione

Quando viene chiamata una funzione:

- Il chiamante (es. `main`) viene temporaneamente **congelato**.
- I parametri attuali (argomenti) vengono **copiati** nei parametri formali.
- La CPU esegue un **salto** all'indirizzo di memoria della funzione.
- Al `return` (o alla fine del blocco), la funzione restituisce il valore e il chiamante si **scongela**, riprendendo dall'istruzione successiva.

? In che ordine vengono eseguite le righe del prossimo programma?

Flusso di Esecuzione — Passo 1/8: Righe 6-7

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

Il programma inizia dal `main`. Vengono allocate e inizializzate le variabili locali `x` e `y`.

Variabili vive:

`main: x = 5, y = 3`



Flusso di Esecuzione — Passo 2/8: Riga 8 (chiamata)

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

Viene invocata `calcola_somma`: il `main` si **congela**, i valori di `x` e `y` vengono **copiati** in `a` e `b`, la CPU salta in memoria.

Variabili vive:

`main`: `x = 5`, `y = 3`, `totale = ?`

Flusso di Esecuzione — Passo 3/8: Riga 1

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

Inizia l'esecuzione di `calcola_somma`: i parametri formali `a` e `b` sono pronti in memoria.

Variabili vive:

`main`: `x = 5`, `y = 3`

`calcola_somma`: `a = 5`, `b = 3`

Flusso di Esecuzione — Passo 4/8: Riga 2

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

Viene dichiarata la variabile locale

`risultato` ed eseguito il calcolo `5 + 3 = 8`.

Variabili vive:

`main: x = 5, y = 3`

`calcola_somma: a = 5, b = 3, risultato = 8`

Flusso di Esecuzione — Passo 5/8: Riga 3

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

`return risultato` restituisce il valore `8`.
Lo Stack Frame di `calcola_somma` viene
distrutto.

Variabili vive:

`main`: `x = 5`, `y = 3`

`calcola_somma`: ~~distrutta~~



Flusso di Esecuzione — Passo 6/8: Riga 8 (ripresa)

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

Il controllo ritorna al `main`, che si **scongela**: il valore restituito `8` viene assegnato a `totale`.

Variabili vive:

`main`: `x = 5`, `y = 3`, `totale = 8`

Flusso di Esecuzione — Passo 7/8: Riga 9

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

Viene stampato il testo:

Totale: 8

Variabili vive:

main: x = 5, y = 3, totale = 8

Flusso di Esecuzione — Passo 8/8: Riga 10

```
1.  int calcola_somma(int a, int b) {  
2.      int risultato = a + b;  
3.      return risultato;  
4.  }  
5.  
6.  int main(void) {  
7.      int x = 5, y = 3;  
8.      int totale = calcola_somma(x, y);  
9.      printf("Totale: %d\n", totale);  
10.     return 0;  
11. }
```

Il `main` termina restituendo `0` al Sistema Operativo (`echo $? ➡ 0`).

Variabili vive:

—

Flusso di Esecuzione: Riepilogo

- Ordine delle righe eseguite: **6, 7, 8, 1, 2, 3, 8, 9, 10.**
- La riga 8 viene "visitata" **due volte**: alla chiamata e alla ripresa (assegnamento a `totale`).

Ispezione live con il debugger (LLDB in VS Code):

Comando	Tasto	Alla riga 8...
Step Over	F10	salta direttamente alla 9, calcolando il risultato
Step Into	F11	entra nella funzione: il cursore va alla riga 1
Step Out	Shift+F11	completa la funzione e torna subito al chiamante

 **Live:** compilare con `-g -Wall`, breakpoint alla riga 8 in VS Code, provare F10 e poi F11 / Shift+F11.

Passaggio per Valore (Copia)

In C il passaggio dei parametri avviene **sempre per valore**: la funzione riceve una **copia del valore**, non la variabile stessa.

- **Parametri formali**: le variabili dichiarate nella firma (es. `int a, int b` in `calcola_somma`). Esistono solo nel corpo della funzione.
- **Parametri attuali (argomenti)**: i valori o le variabili passati alla chiamata (es. `x, y` o `5, 3` nel `main`).

 Qualsiasi modifica al parametro formale **non ha alcun effetto** sulla variabile originale del chiamante.



Passaggio per Valore: Dimostrazione

```
#include <stdio.h>

void incrementa(int n) {
    n = n + 1; // Modifica solo la copia locale 'n'
    printf("Valore dentro la funzione: %d\n", n);
}

int main(void) {
    int x = 10;
    incrementa(x); // Passiamo x (parametro attuale) alla funzione
    printf("Valore nel main dopo la chiamata: %d\n", x); // x rimarrà 10!
    return 0;
}
```

? Cosa stampano le due `printf`?

Passaggio per Valore: Output

```
gcc -Wall valore.c -o valore && ./valore
```

```
Valore dentro la funzione: 11  
Valore nel main dopo la chiamata: 10
```

- `n` è una **copia** di `x`, in un'altra zona di memoria (lo Stack Frame di `incrementa`).
- `n` diventa `11`, ma `x` nel `main` resta `10`.

 **Live:** eseguire con il debugger, Step Into su `incrementa(x)` e osservare `n` e `x` nel pannello Variables.

Scope Locale (Block Scope)

- Le variabili dichiarate in una funzione (**parametri formali compresi**) nascono alla chiamata e vengono **distrutte** quando la funzione termina.
- Variabili con lo **stesso nome** in funzioni diverse **non collidono**: risiedono in aree di memoria differenti.

```
void funzioneA(void) {  
    int valore = 42; // Locale a funzioneA  
    printf("Dentro funzioneA: %d\n", valore);  
}  
  
void funzioneB(void) {  
    int valore = 99; // Locale a funzioneB. Non collide con quella di funzioneA!  
    printf("Dentro funzioneB: %d\n", valore);  
}
```



Scope Locale: Esempio Completo

```
#include <stdio.h>

void funzioneA(void) {
    int valore = 42;
    printf("Dentro funzioneA: %d\n", valore);
}

void funzioneB(void) {
    int valore = 99;
    printf("Dentro funzioneB: %d\n", valore);
}

int main(void) {
    funzioneA();
    funzioneB();
    // printf("%d", valore); // ERRORE DI COMPILAZIONE!
    return 0;                // 'valore' non esiste nel main
}
```

Dentro funzioneA: 42
Dentro funzioneB: 99

 **Live:** decommentare la `printf` nel `main` e compilare con `gcc -Wall`: leggere l'errore (`'valore' undeclared`).

Variabili Globali: l'Antipattern

```
#include <stdio.h>

int contatore = 0; // VARIABILE GLOBALE - Sconsigliata!

void incrementa_contatore(void) {
    contatore++; // Modifica lo stato globale
}

void resetta_contatore(void) {
    contatore = 0; // Modifica lo stato globale inaspettatamente
}

int main(void) {
    incrementa_contatore();
    incrementa_contatore();
    printf("Contatore: %d\n", contatore); // Stampa 2

    resetta_contatore(); // Se chiamata altrove, lo stato si corrompe

    printf("Contatore dopo reset: %d\n", contatore); // Stampa 0!
    return 0;
}
```

Contatore: 2
Contatore dopo reset: 0

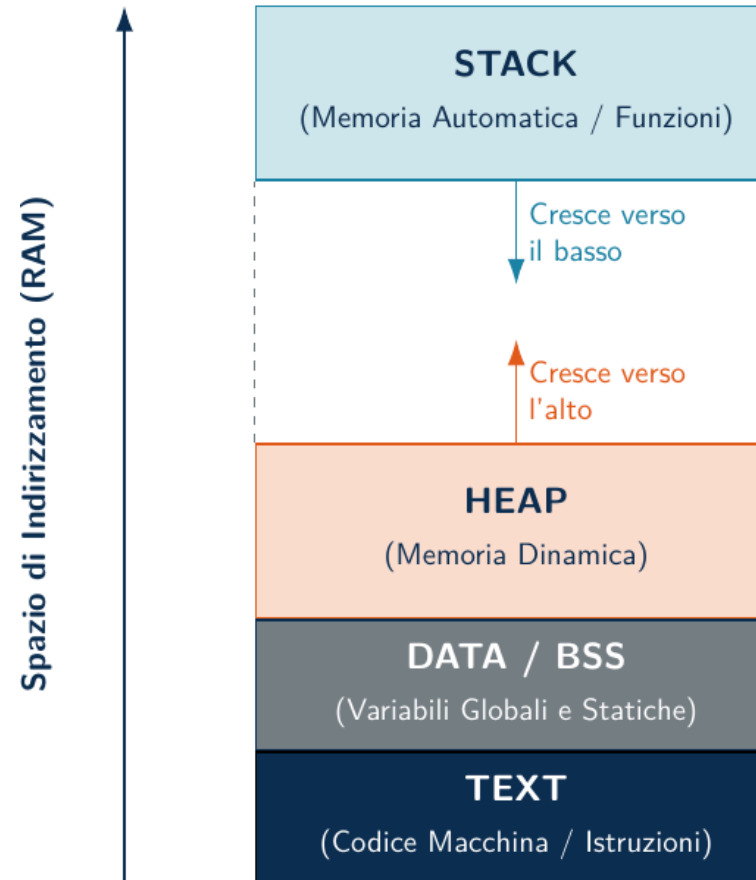
- Dichiarate **fuori** da qualsiasi funzione: accessibili (e modificabili) **da chiunque**.
- Rompono l'**incapsulamento**: lo stato diventa imprevedibile e difficile da debuggare.
-  **Vivamente sconsigliate.**

Sotto il Cofano: Organizzazione della RAM

La RAM di un processo è divisa in segmenti:

- **Text:** istruzioni CPU (codice macchina).
- **Data / BSS:** variabili globali e statiche (Data: inizializzate, es. `int x = 42;`; BSS: non inizializzate, azzerate dal SO all'avvio).
- **Heap:** allocazione dinamica (manuale).
- **Stack:** struttura LIFO (*Last In, First Out*) gestita automaticamente per le funzioni.

Lo Stack cresce verso indirizzi bassi (Push) e si contrae al return (Pop).



Lo Stack Frame (Record di Attivazione)

Ogni chiamata crea nello Stack un blocco contiguo, lo **Stack Frame**, che contiene:

- i parametri passati alla funzione;
- l'**indirizzo di ritorno** (l'istruzione del chiamante a cui saltare alla fine);
- il puntatore al **vecchio frame** (per ripristinare lo stato precedente);
- le **variabili locali** della funzione.

Al `return` il frame viene rimosso (**Pop**) e lo spazio è subito invalidato, disponibile per le chiamate successive.

➡ Ecco perché le variabili locali **non sopravvivono** al termine della funzione.



Allineamento e Padding a 16 Byte

- Sulle architetture a 64 bit i dati viaggiano tra RAM e CPU su bus ottimizzati per blocchi grandi.
- Per efficienza, ogni Stack Frame è allineato a multipli di **16 byte**.
- Se lo spazio di variabili locali e parametri non è multiplo di 16, il compilatore aggiunge byte vuoti invisibili: il **padding**.

Esempio: una sola variabile locale `int` (4 byte) ➡ il frame viene comunque allineato a 16 byte con **12 byte di padding**.

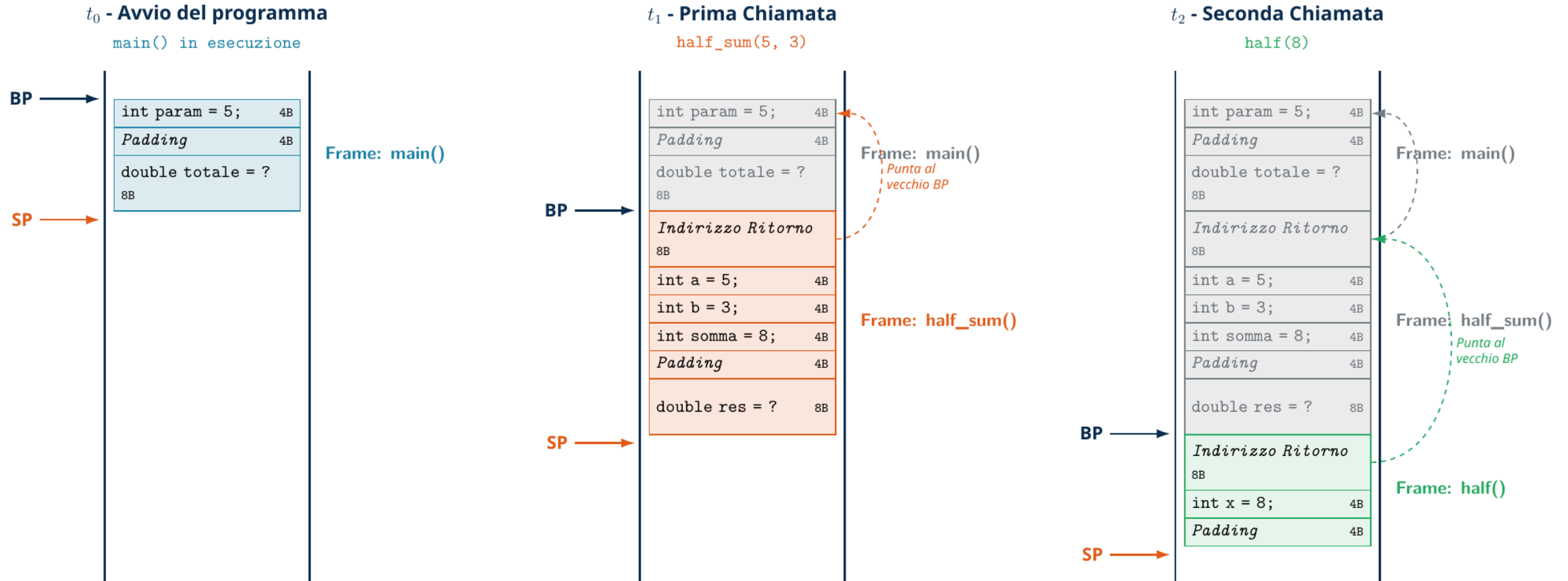
? Vediamolo su un programma con tre funzioni che si chiamano a catena.

⚙ Stack Frames: il Codice d'Esempio

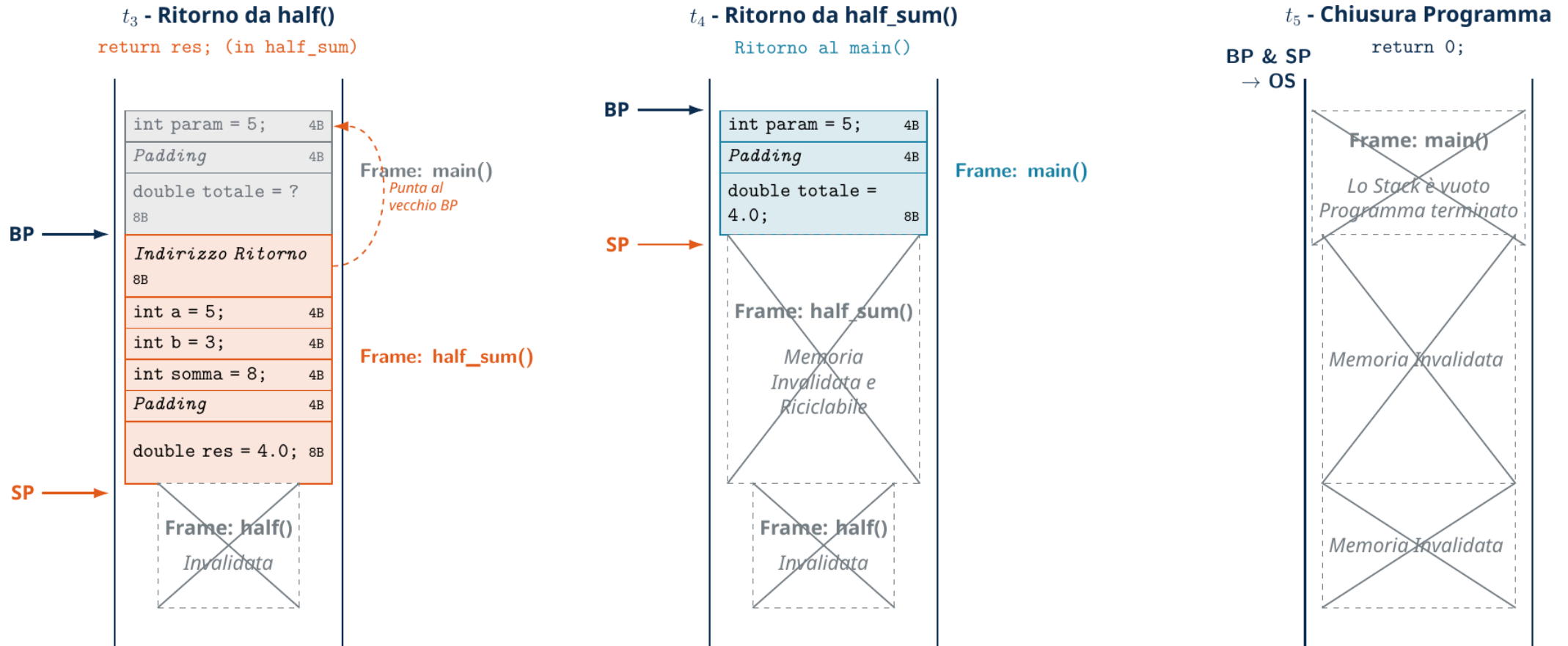
```
double half(int x) {  
    return (double)x / 2;  
}  
  
int half_sum(int a, int b) {  
    int somma = a + b;  
    double res = half(somma);  
    return res;  
}  
  
int main(void) {  
    int param = 5;  
    double totale = half_sum(param, 3);  
    return 0;  
}
```

- Catena di chiamate: `main()` ➡ `half_sum(5, 3)` ➡ `half(8)` ➡ ritorno a `half_sum`
➡ ritorno a `main`.

⚙ Stack Frames: Espansione (Push)



⚙ Stack Frames: Contrazione (Pop)





Come Fa la CPU a Sapere Quanta Memoria Serve?

Lo decide il **compilatore**, a tempo di compilazione, grazie al **sistema dei tipi**:

- conosce a priori la dimensione di ogni variabile locale e parametro (`int` = 4 byte, `double` = 8 byte);
- somma lo spazio occupato nel corpo di ciascuna funzione;
- calcola l'allineamento a 16 byte, aggiungendo il padding necessario;
- genera le istruzioni macchina per spostare lo Stack Pointer, ad esempio:

```
sub rsp, 24
```

Navigare lo Stack: RSP e RBP

La CPU tiene traccia del frame attivo con due registri:

- **RSP (Stack Pointer)**: punta sempre alla **cima** dello Stack (l'indirizzo occupato più basso). Si sposta a ogni push o pop.
- **RBP (Base Pointer)**: punta a un indirizzo **fisso**, la base del frame corrente.
- Variabili locali e parametri sono referenziati con un **offset** rispetto a RBP (es. `RBP - 4`).

Al `return`, la CPU ripristina RSP e RBP del chiamante e salta all'indirizzo di ritorno.

? Come appare tutto questo in Assembly per una semplice `somma(2, 3)` ?



Esempio x86-64: il Codice C

```
// Codice C
int somma(int a, int b) {
    return a + b;
}

int main(void) {
    int ris = somma(2, 3);
    return 0;
}
```

- Due argomenti in ingresso, un valore di ritorno, una variabile locale `ris` nel `main`.

Esempio x86-64: l'Assembly Commentato

```
; Assembly Intel x86-64 corrispondente
main:
    ; 1. Preparazione dei parametri negli appositi registri
    mov edi, 2      ; Metti il primo argomento (2) nel registro EDI
    mov esi, 3      ; Metti il secondo argomento (3) nel registro ESI

    ; 2. Invocazione della funzione
    call somma      ; Salva l'indirizzo della prossima istruzione nello Stack
                    ; e salta all'etichetta "somma"

    ; ... La CPU ritorna esattamente qui al termine della funzione ...
    mov DWORD PTR [rbp-4], eax ; Salva il valore restituito in EAX dentro 'ris'
    ret

somma:
    ; 3. Setup (Prologo) del nuovo Stack Frame
    push rbp        ; Salva il Base Pointer del chiamante nello Stack
    mov rbp, rsp    ; Imposta il nuovo Base Pointer pari allo Stack Pointer corrente

    ; 4. Esecuzione del calcolo
    add edi, esi     ; Somma a + b (EDI + ESI)
    mov eax, edi     ; Memorizza il risultato nel registro EAX (registro di ritorno)

    ; 5. Distruzione (Epilogo) dello Stack Frame
    pop rbp         ; Ripristina il vecchio Base Pointer del chiamante dallo Stack
    ret             ; Preleva l'indirizzo di ritorno dallo Stack e vi salta indietro
```



Verifica con Python Tutor

- Su pythontutor.com, impostando il linguaggio **C (GCC)**, si può eseguire il codice passo-passo.
- Si vede graficamente la **pila dei record di attivazione**: chiamate, ciclo di vita delle variabili locali, ritorno al chiamante.

 **Live:** incollare su Python Tutor il programma `main` ➡ `half_sum` ➡ `half` e avanzare riga per riga, mostrando i frame che si impilano e poi scompaiono.

Cosa Abbiamo Imparato Oggi

- Le funzioni eliminano la duplicazione del codice (principio **DRY**) e danno **astrazione**.
- **Firma** (tipo di ritorno, nome, parametri) vs **corpo**; prototipi in cima, definizioni in fondo.
- Il C passa i parametri **esclusivamente per valore** (copia).
- Le variabili locali vivono nello **Stack Frame** della funzione e muoiono al `return`.
Niente globali!
- Frame allineati a **16 byte** con padding, dimensioni calcolate dal compilatore.
- I registri `RSP` e `RBP` tracciano cima e base del frame attivo.

 **Per casa:** esercizi della Lezione 7 sul sito del corso.

Grazie per l'attenzione!

Sito Web del Corso: antoninofurnari.github.io/prog1-2627