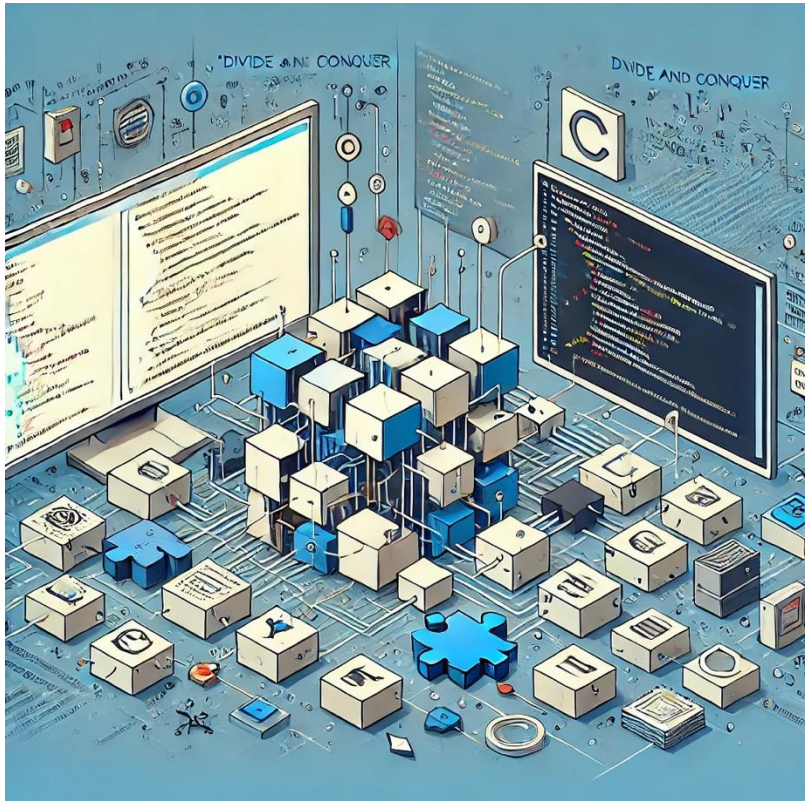
Le funzioni

Introduzione alle funzioni

- I programmi reali sono spesso molto più grandi di quelli introdotti inizialmente
- È importante sviluppare e mantenere programmi di grandi dimensioni in modo efficiente
- Strategia efficace: **"dividi et impera"**:
  - Suddividere il programma in elementi più piccoli e gestibili



Fonte: ChatGPT

Modularizzare i programmi in C

- Utilizzo delle funzioni per creare programmi modulari
- Combinazione di nuove funzioni con quelle della **Libreria Standard del C**
  - Calcoli matematici
  - Manipolazioni di stringhe e caratteri
  - Input/Output e altre operazioni comuni
- Funzioni predefinite facilitano lo sviluppo, fornendo capacità necessarie preconfenzionate
- Funzioni come `printf, scanf e pow` fanno parte della Libreria Standard.
- Alcune parti della Libreria sono **opzionali** e potrebbero non essere disponibili in tutti i compilatori

Riutilizzo del codice

- Evitate di **"reinventare la ruota"**
- Familiarizzare con la Libreria Standard per ridurre i tempi di sviluppo
- Usate le funzioni standard quando possibile:
  - Sono scritte da esperti, ben testate, efficienti e migliorano la portabilità dei programmi
- È possibile definire funzioni per eseguire compiti specifici che possono essere utilizzate in più punti di un programma
- Le istruzioni che definiscono la funzione vengono scritte una volta e sono nascoste alle altre funzioni
- Tale occultamento è fondamentale per una buona ingegneria del software

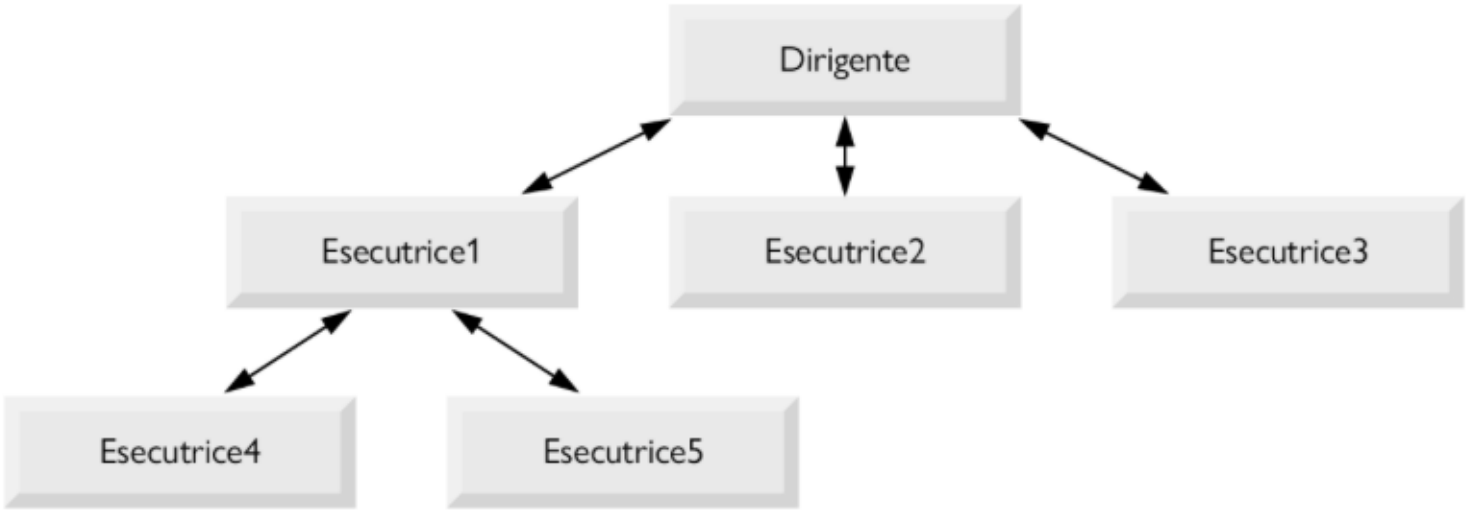
Chiamata di funzioni

- Una funzione è invocata tramite una **chiamata di funzione**
- La chiamata specifica il **nome** della funzione e fornisce gli **argomenti** necessari per eseguire il compito
- La funzione **chiamante** (dirigente) richiede a una funzione **chiamata** (esecutrice) di svolgere un compito
- La funzione esecutrice esegue il compito e ritorna alla funzione chiamante al completamento

  Esempio: `int printf(const char *format, ...)`
- La funzione chiamante non conosce i dettagli di come la funzione esecutrice esegue il compito

Gerarchia del codice

- Le funzioni esecutrici possono a loro volta chiamare altre funzioni
- La funzione dirigente può rimanere ignara delle chiamate effettuate dalle funzioni esecutrici
- Un diagramma può mostrare la comunicazione gerarchica tra una funzione dirigente e diverse funzioni esecutrici



Fonte: Deitel & Deitel

Libreria `math.h`

- Permette di eseguire calcoli matematici comuni
- Esempio: Calcolo e stampa della radice quadrata di 900.0

```
printf("%.2f", sqrt(900.0)); // Stampa "30.00"
```

- La funzione `sqrt` riceve un argomento di tipo `double` e restituisce un risultato di tipo `double`
- Tutte le funzioni della libreria `math` che restituiscono valori in virgola mobile restituiscono `double`
- I valori double possono essere stampati con la specifica di conversione `%f`

Risultati e argomenti

- Il risultato di una funzione può essere memorizzato in una variabile:

```
double result = sqrt(900.0);
```

- **Argomenti** delle funzioni:
  - Possono essere costanti, variabili o espressioni
- Esempio con espressione:

```
double c = 13.0, d = 3.0, f = 4.0;
printf("%.2f", sqrt(c + d * f)); // Stampa "5.00»
```

Libreria `math.h`

Funzione	Descrizione	Esempio
`sqrt(x)`	radice quadrata di *x*	`sqrt(900.0)` è uguale a `30.0` `sqrt(9.0)` è uguale a `3.0`
`cbrt(x)`	radice cubica di *x* (solo per il C99 e il C11)	`cbrt(27.0)` è uguale a `3.0` `cbrt(-8.0)` è uguale a `-2.0`
`exp(x)`	funzione esponenziale e^x	`exp(1.0)` è uguale a `2.718282` `exp(2.0)` è uguale a `7.389056`
`log(x)`	logaritmo naturale di *x* (in base *e*)	`log(2.718282)` è uguale a `1.0` `log(7.389056)` è uguale a `2.0`

`log10(x)`	logaritmo di *x* (in base 10)	`log10(1.0)` è uguale a `0.0`
		`log10(10.0)` è uguale a `1.0`
		`log10(100.0)` è uguale a `2.0`

`fabs(x)`	valore assoluto di *x* come numero in virgola mobile	`fabs(13.5)` è uguale a `13.5`
		`fabs(0.0)` è uguale a `0.0`
		`fabs(-13.5)` è uguale a `13.5`

Fonte: Deitel & Deitel

Libreria `math.h`

`ceil(x)`	arrotonda *x* all'intero più piccolo non minore di *x*	`ceil(9.2)` è uguale a `10.0` `ceil(-9.8)` è uguale a `-9.0`
`floor(x)`	arrotonda *x* all'intero più grande non maggiore di *x*	`floor(9.2)` è uguale a `9.0` `floor(-9.8)` è uguale a `-10.0`
`pow(x, y)`	*x* elevato alla potenza *y* (x^y)	`pow(2, 7)` è uguale a `128.0` `pow(9, .5)` è uguale a `3.0`
`fmod(x, y)`	resto di *x*/*y* come numero in virgola mobile	`fmod(13.657, 2.333)` è uguale a `1.992`
`sin(x)`	funzione trigonometrica seno di *x* (*x* in radianti)	`sin(0.0)` è uguale a `0.0`
`cos(x)`	funzione trigonometrica coseno di *x* (*x* in radianti)	`cos(0.0)` è uguale a `1.0`
`tan(x)`	funzione trigonometrica tangente di *x* (*x* in radianti)	`tan(0.0)` è uguale a `0.0`

Fonte: Deitel & Deitel

Funzioni in C

- Le funzioni permettono di suddividere un programma in **moduli** più gestibili
- Nei programmi complessi, la funzione `main` è spesso composta da una serie di chiamate a funzioni
- Queste funzioni chiamate eseguono la maggior parte del lavoro del programma

Perché «funzionalizzare» un programma?

- **Divide et impera (Dividi e Conquista)**: Semplifica lo sviluppo del programma suddividendolo in parti più gestibili
- **Riutilizzo del software**:
  - Creazione di nuovi programmi utilizzando funzioni esistenti.
  - Concetto chiave nei linguaggi orientati agli oggetti derivati dal C (C++, Java).
- **Astrazione:**
  - Uso di funzioni standard per compiti specifici (Esempio: `printf, scanf, pow`)

Perché «funzionalizzare» un programma?

- **Evitare la ripetizione del codice**:
  - Racchiudere il codice in funzioni permette di richiamarlo in diversi punti del programma
- **Buone pratiche per le funzioni:**
  - Ogni funzione dovrebbe eseguire *un unico compito* ben definito
  - Il *nome* della funzione deve riflettere chiaramente il compito svolto
  - Se una funzione svolge troppi compiti, suddividerla in funzioni più piccole (*scomposizione*)

Funzione square

```c
// fig05_01.c
// Creazione e uso di una funzione.
#include <stdio.h>

int  square( int  number); // prototipo di funzione

int main(void) {
   // ripeti 10 volte e ogni volta calcola e stampa il quadrato di x
   for (int x = 1; x <= 10; ++x) {
      printf("%d  ", square(x) ); // chiamata della funzione
   }

    puts("");
 }

 // la definizione di square restituisce il quadrato del suo parametro
 int  square( int  number) {  // number e' una copia dell'argomento della funzione
     return  number * number;  // restituisce il quadrato di number come valore int
 }
```

```
1  4  9  16  25  36  49  64  81  100
```

Fonte: Deitel & Deitel

Funzione `square`

```
int main(void) {
   // ripeti 10 volte e ogni volta calcola e stampa il quadrato di x
   for (int x = 1; x <= 10; ++x) {
      printf("%d  ", square(x)); // chiamata della funzione
   }
```

- La funzione `square` è chiamata nella funzione `main` all'interno di `printf`
- Riceve una copia del valore dell'argomento `x` nel parametro `number`
- Calcola `number * number` e restituisce il risultato
- Il risultato è passato a `printf`, che lo stampa sullo schermo.
- L'intero processo è ripetuto 10 volte tramite un ciclo `for`

Chiamata di funzione `square`

```
// la definizione di square restituisce il quadrato del suo parametro
int  square( int  number) {  // number e' una copia dell'argomento della funzione
    return  number * number;  // restituisce il quadrato di number come valore int
}
```

- Accetta un parametro **intero** chiamato `number`
- La parola chiave `int` indica che la funzione restituisce un valore intero
- L'istruzione `return` restituisce il quadrato di number (ossia `number * number`)

Definire una funzione

- Scegliere nomi chiari per funzioni e parametri
- Migliora la leggibilità del codice e riduce la necessità di commenti esplicativi
- Scrivi programmi come collezioni di piccole funzioni (modularità)
  - Facilita la scrittura, il debugging, la manutenzione, la modifica e il riutilizzo del codice

Definire una funzione

- Funzioni con molti parametri possono essere troppo complesse
- Meglio suddividere funzioni grandi in più funzioni più piccole e specializzate
- Dichiarazione del tipo di ritorno, nome e lista dei parametri dovrebbe essere su una singola riga, se possibile, per mantenere il codice chiaro e compatto

Quiz

Variabili e parametri

- Le variabili definite all'interno di una funzione sono **variabili locali**
- Vi si può accedere solo all'interno della funzione in cui sono state dichiarate
- Le funzioni utilizzano **parametri** per ricevere dati
- Gli **argomenti** passati durante la chiamata della funzione permettono la comunicazione tra le funzioni
- I parametri di una funzione sono considerati **variabili locali** della funzione stessa
  - Sono visibili e utilizzabili solo all'interno della funzione in cui sono definiti

Parametri e argomenti

- I parametri sono le variabili definite nella dichiarazione di una funzione o metodo
- Servono come "posti riservati" per i dati che la funzione utilizzerà.
  - I parametri sono specificati nella **dichiarazione** e **definizione** della funzione
  - Rappresentano le **specifiche** di cosa una funzione si aspetta di ricevere quando viene chiamata
- Gli argomenti sono i valori **effettivi** che vengono passati alla funzione quando la chiamata viene effettuata.
  - Gli argomenti sono forniti durante la **chiamata** della funzione
  - Rappresentano i dati concreti che verranno utilizzati dalla funzione

Parametri e argomenti

```
// la definizione di square restituisce il quadrato del suo parametro
int  square( int  number) {  // number e' una copia dell'argomento della funzione
    return  number * number;  // restituisce il quadrato di number come valore int
}
```

```
 ...

int main(void) {
   // ripeti 10 volte e ogni volta calcola e stampa il quadrato di x
   for (int x = 1; x <= 10; ++x) {
      printf("%d  ", square(x) ); // chiamata della funzione
   }
```

- Parametro: variabile `int` di nome `number`
- Argomento: variabile `x`

Prototipo di una funzione

```
int square(int number);
```

- Il prototipo informa il compilatore sui dettagli della funzione:
  - `int` prima del nome indica il tipo di valore restituito dalla funzione
  - `int` dentro le parentesi indica che la funzione accetta un parametro di tipo intero
- Il punto e virgola alla fine di un prototipo di funzione è obbligatorio
  - **Dimenticarlo** è considerato un errore di sintassi
- Il compilatore confronta la chiamata della funzione con il suo prototipo per garantire:
  - **Numero di argomenti** corretto
  - **Tipi degli argomenti** corretti
  - **Ordine degli argomenti** corretto
  - **Tipo di ritorno** coerente con l'uso previsto

Coerenza tra prototipo e definizione

- Il prototipo della funzione, la definizione e le chiamate devono:
  - Concordare nel **numero**, **tipo** e **ordine** degli argomenti e dei parametri
  - Avere lo stesso **tipo di ritorno**
- Il tipo di ritorno influisce sull'uso della funzione:
  - Funzioni con tipo di ritorno **void** non possono essere usate per **assegnazioni:**
  - Esempio:

```
// Definizione della funzione di tipo void
void printSum(int a, int b) {
// Calcolo della somma e stampa del risultato
int sum = a + b;
printf("La somma di %d e %d è %d\n", a, b, sum);
}
```

Formato definizione di una funzione

```
tipo-del-valore-di-ritorno[1] nome-della-funzione[2](lista-dei-parametri[3]){

istruzioni

}
```

[1] **Tipo del valore di ritorno:** Indica il tipo del valore che la funzione restituisce

[2] **Nome della funzione:** Qualsiasi identificatore valido che rappresenta la funzione

[3] **Lista dei parametri:** Elenco dei parametri che la funzione accetta, separati da virgole. Se non ci sono parametri, usa `void` nella lista dei parametri

Parametri e tipo di ritorno

- Ogni parametro deve specificare il suo tipo. Mancare di farlo porta a **errori di compilazione**
- La dichiarazione di `void` nella lista dei parametri indica che la funzione non accetta parametri
- **Errori comuni:**
  - **Punto e virgola errato:** Non inserire un punto e virgola dopo la parentesi destra che chiude la lista dei parametri nella definizione della funzione
  - **Nomi ambigui:** Evitare di usare lo stesso nome per variabili locali e per i parametri della funzione per evitare ambiguità

Corpo della funzione

- Le istruzioni racchiuse tra parentesi graffe {} formano il **corpo della funzione** (anche chiamato blocco).
- **Variabili locali** possono essere dichiarate all'interno di qualsiasi blocco e i blocchi possono essere **annidati**
- Le **funzioni non possono essere annidate**: non è permesso definire una funzione all'interno di un'altra funzione (*errore di sintassi*)

Restituzione del controllo da una funzione

- Il controllo viene restituito quando si raggiunge la parentesi graffa che termina la funzione, oppure con l'istruzione `return`
- La funzione può restituire il controllo con o senza valore di ritorno
- Funzione `main`
  - **Valore di ritorno:** `int` indica lo stato di esecuzione del programma
  - **Valore di successo:** `return 0;`
    (0 solitamente indica che il programma è stato eseguito correttamente)
- **Comportamento standard:** Il C standard imposta che `main` restituisca implicitamente 0 se non viene specificata una istruzione `return`
- **Valori di errore:** È possibile restituire valori diversi da zero per segnalare problemi durante l'esecuzione

Esempio: funzione massimo

```c
// fig05_02.c
// Trovare il maggiore di tre interi.
#include <stdio.h>

int maximum( int x, int y, int z); // prototipo di funzione

int main(void) {
   int number1 = 0; // primo intero inserito dall'utente
   int number2 = 0; // secondo intero inserito dall'utente
   int number3 = 0; // terzo intero inserito dall'utente

   printf("%s", "Enter three integers: ");
    scanf("%d%d%d", &number1, &number2, &number3);

    // number1, number2 e number3 sono argomenti
    // nella chiamata della funzione maximum
    printf("Maximum is: %d\n", maximum(number1, number2, number3) );
}

// Definizione della funzione maximum
int maximum( int x, int y, int z) {
    int max = x; // supponi che x sia il maggiore

    if (y > max) { // se y e' piu' grande di max,
      max = y; // assegna y a max
    }

    if (z > max) { // se z e' piu' grande di max,
      max = z; // assegna z a max
    }

    return max; // max e' il valore piu' grande
}
```

```
Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 47 32 14
Maximum is: 47

Enter three integers: 35 8 79
Maximum is: 79
```

Fonte: Deitel & Deitel

Prototipi di funzione

- Il prototipo di funzione è una caratteristica fondamentale del C, **presa in prestito dal C++**
- I prototipi aiutano il compilatore a verificare la correttezza delle chiamate alle funzioni
- Le versioni precedenti del C non eseguivano controlli sui prototipi
- Le chiamate alle funzioni potevano essere fatte in modo **improprio** senza che il compilatore segnalasse errori
- Questo poteva causare errori critici o problemi difficili da rilevare durante l'esecuzione.
- I prototipi di funzione correggono queste carenze e **migliorano la sicurezza del codice.**
- È importante includere i prototipi per tutte le funzioni per sfruttare il controllo dei tipi del C
    - Utilizzare la direttiva #include per inserire prototipi da file di intestazione della Libreria Standard, librerie di terze parti, o intestazioni personalizzate

Prototipi di funzione

```
int maximum(int x, int y, int z);
```

- La funzione maximum riceve tre argomenti di tipo `int` e restituisce un valore di tipo `int`
- I nomi dei parametri nel prototipo sono per **documentazione**
- Il compilatore ignora i nomi dei parametri. Ad esempio, anche il seguente prototipo è valido:

```
int maximum(int , int , int);
```

Errori di compilazione

- Una chiamata a una funzione che non corrisponde al prototipo della funzione genera un errore di compilazione
- Un errore di compilazione si verifica anche se ci sono discrepanze tra il prototipo e la definizione della funzione
- **Esempio di errore**:
  - Prototipo:

```
void maximum(int x, int y, int z);
```

  - Definizione:

```
int maximum(int x, int y, int z){
// istruzioni
}
```

- l compilatore genererebbe un errore poiché il tipo di ritorno `void` nel prototipo è diverso dal tipo di ritorno `int` nella definizione

Coercizione degli argomenti

- La coercizione degli argomenti è la **conversione implicita** degli argomenti al tipo appropriato definito nel prototipo della funzione
- La funzione `sqrt` della libreria `math` accetta un parametro di tipo `double`, ma può essere chiamata con un argomento di tipo `int`
- Il compilatore converte **automaticamente** l'argomento `int` in `double`
  - Esempio: `printf("%.3f\n", sqrt(4));`
  - Stampa 2.000 perché sqrt(4) converte 4 da int a double (4.0) e calcola la radice quadrata correttamente

Regole di conversione aritmetica

- Le conversioni implicite devono seguire le **normali regole di conversione aritmetica** del C, che determinano come i valori possono essere convertiti tra tipi **senza perdita di dati**
- Conversione da `int` a `double`: non causa perdita di dati perché `double` può rappresentare un intervallo di valori molto più grande di `int`
- Conversione da `double` a `int`: la parte frazionaria del double viene troncata, modificando il valore originale

Espressioni con tipi misti

- Le regole di conversione aritmetica sono gestite dal **compilatore** e si applicano a espressioni con tipi misti, ovvero espressioni che contengono valori di più tipi di dati
- Il compilatore crea *copie temporanee* dei valori da convertire e li converte al tipo di dato "più alto" nell'espressione, processo noto come **promozione**
- **Espressioni con tipi misti e valori in virgola mobile:**
  - `long double`: gli altri valori vengono convertiti in `long double`
  - `double`: gli altri valori vengono convertiti in `double`
  - `float`: gli altri valori vengono convertiti in `float`
- **Espressioni con solo tipi interi**
  - Se l'espressione contiene solo tipi interi, le normali regole di conversione aritmetica stabiliscono un insieme di regole di **promozione intera**

Specifiche di conversione per `printf` e `scanf` (tipi in virgola mobile)

Tipo di dati	Specifica di conversione per *printf*	Specifica di conversione per *scanf*
Tipi in virgola mobile		
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`

Fonte: Deitel & Deitel

Specifiche di conversione per `printf` e `scanf` (tipi interi)

Tipi interi		
`unsigned long long int`	`%llu`	`%llu`
`long long int`	`%lld`	`%lld`
`unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`unsigned short`	`%hu`	`%hu`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

Fonte: Deitel & Deitel

Conversioni tra tipi di dati

- È possibile convertire un valore in un tipo più basso solo tramite:
  - assegnazione esplicita a una variabile di tipo più basso
  - utilizzo di un operatore di `cast`
- Esempio:
  - se passiamo un `double` alla funzione `square` che accetta un `int`, il `double` viene convertito in `int`, portando a risultati non corretti
  - `square(4.5)` restituisce 16 invece di 20.25
- Convertire da un tipo di dato **più alto a uno più basso** può cambiare il valore dei dati
- Molti compilatori emettono **avvisi** quando si verificano queste conversioni

Nota sui prototipi di funzione

- Se non è presente un prototipo per una funzione, il compilatore **crea** un prototipo basato sulla prima occorrenza della funzione (definizione o chiamata).
  - questo può generare avvisi o errori a seconda del compilatore.
- Includete sempre i prototipi per le funzioni definite o usate nel programma per prevenire errori e avvisi in fase di compilazione
- Un prototipo di funzione posto al di fuori della definizione di qualsiasi altra funzione è valido per tutte le chiamate alla funzione che seguono il prototipo
- Un prototipo posto all'interno del corpo di una funzione è valido solo per le chiamate a quella funzione fatte dopo il prototipo

Intervallo

Fonte: PlaygroundAI

Pila delle chiamate delle funzioni

- Il C esegue le chiamate di funzione usando il concetto di pila
- Una pila è una struttura di dati che segue il principio **Last-In, First-Out (LIFO)**
  - l'ultimo elemento inserito è il primo a essere rimosso.
- Operazioni Fondamentali:
  - **`push`**: aggiungere un elemento in cima alla pila.
  - **`pop`**: rimuovere l'elemento in cima alla pila.

Fonte: Wikipedia

Pila delle chiamate delle funzioni

- La pila delle chiamate delle funzioni (o **pila di esecuzione del programma**) gestisce le chiamate e i ritorni delle funzioni
- Supporta anche la creazione, il mantenimento e la distruzione delle variabili locali delle funzioni chiamate
- **Chiamata di funzione:**
  - Quando una funzione viene chiamata, i dati relativi alla chiamata (come gli argomenti e il punto di ritorno) vengono inseriti nella cima della pila
- **Ritorno dalla funzione:**
  - Quando una funzione termina, i dati relativi alla chiamata vengono rimossi dalla cima della pila e il controllo ritorna alla funzione chiamante

Record di attivazione

- Una funzione può chiamare altre funzioni, e queste possono a loro volta chiamare ulteriori funzioni
- Tutte le chiamate devono essere gestite prima che la funzione ritorni alla sua funzione chiamante
- La pila delle chiamate tiene traccia degli **indirizzi di ritorno** necessari per le funzioni per tornare alla funzione che le ha chiamate
- Ogni volta che una funzione chiama un'altra funzione, viene inserito un nuovo elemento nella cima della pila, chiamato **record di attivazione**
- Questo record contiene:
  - L'indirizzo di ritorno per tornare alla funzione chiamante
  - Altre informazioni aggiuntive, come gli argomenti passati e le variabili locali

Gestione della memoria

- La memoria disponibile in un computer è finita
- Solo una certa quantità di memoria può essere utilizzata per memorizzare i record di attivazione nella pila delle chiamate delle funzioni
- Se il numero di chiamate di funzioni supera la capacità della pila delle chiamate, si verifica un errore noto come **stack overflow**
- Questo errore avviene quando non c'è più spazio nella pila per memorizzare nuovi record di attivazione
- Il nome **stackoverflow.com** di un popolare sito web per la programmazione deriva da questo concetto

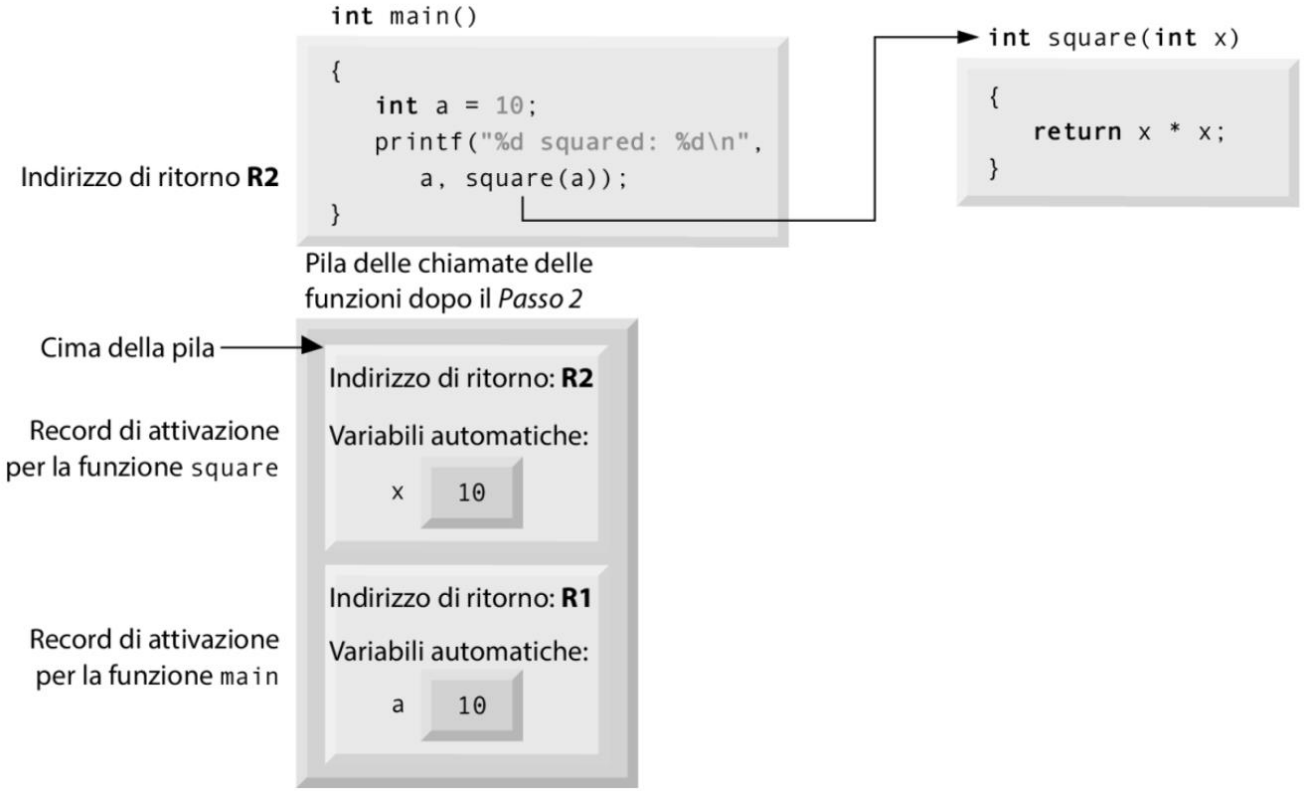
Esempio: funzione `square`

- Passo 1: il sistema operativo invoca il main per eseguire l'applicazione

Fonte: Deitel & Deitel

Esempio: funzione `square`

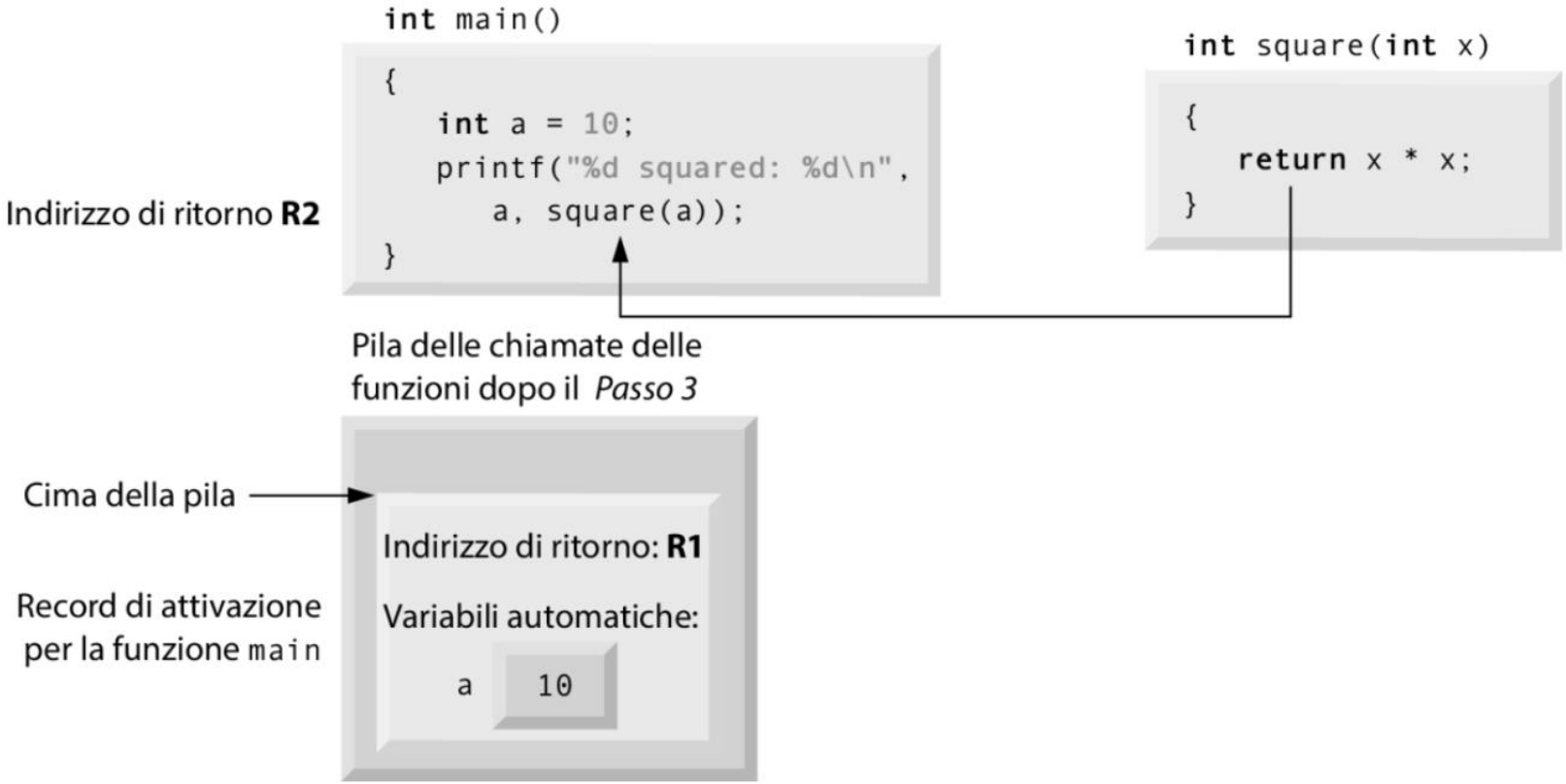
- Passo 2: main invoca la funzione square per eseguire il calcolo



Fonte: Deitel & Deitel

Esempio: funzione `square`

- Passo 3: square restituisce il suo risultato a main



Fonte: Deitel & Deitel

Ordine di esecuzione

- Anche `printf` è una funzione, ma gli argomenti devono essere noti prima dell'esecuzione
- Potreste essere propensi a dire che `printf` chiami `square`
- **In realtà:**
  1. Il sistema operativo chiama `main`, quindi il record di attivazione di `main` viene inserito con un `push` in cima alla pila
  2. `main` chiama `square`, quindi il record di attivazione di `square` viene inserito con un `push` in cima alla pila
  3. `square` esegue il calcolo e restituisce a `main` un valore da usare nella lista di argomenti di `printf`, quindi il record di attivazione di `square` viene rimosso dalla pila con un `pop`
  4. `main` chiama `printf`, quindi il record di attivazione di `printf` viene inserito con un `push` in cima alla pila
  5. `printf` stampa i suoi argomenti, poi restituisce il suo risultato a `main`, quindi il record di attivazione di `printf` viene rimosso dalla pila con un `pop`
  6. `main` termina, quindi il record di attivazione di `main` viene rimosso dalla pila con un `pop`

File di intestazione (header)

- Ogni libreria standard ha un file di intestazione corrispondente
- **Prototipi** di funzioni per tutte le funzioni della libreria
- **Definizioni** di vari tipi di dati e costanti necessari per le funzioni

`<assert.h>`	Contiene informazioni per aggiungere istruzioni di diagnostica che agevolano il debugging (ricerca di errori e correzione) dei programmi.
`<ctype.h>`	Contiene prototipi di funzioni per le funzioni che verificano certe proprietà dei caratteri e per le funzioni che si possono usare per convertire lettere minuscole in lettere maiuscole e viceversa.
`<float.h>`	Contiene i limiti per i valori in virgola mobile del sistema.
`<limits.h>`	Contiene i limiti per i valori interi del sistema.
`<math.h>`	Contiene i prototipi di funzioni per le funzioni della libreria math.
`<signal.h>`	Contiene i prototipi di funzioni e le macro per gestire varie situazioni che possono insorgere durante l'esecuzione di un programma.
`<stdarg.h>`	Definisce le macro per trattare liste di argomenti per una funzione, il cui numero e i cui tipi non sono noti a priori.
`<stdio.h>`	Contiene i prototipi di funzioni per le funzioni della libreria standard di input/output, nonché le informazioni usate da queste.
`<stdlib.h>`	Contiene i prototipi di funzioni per convertire numeri in testo e testo in numeri, per allocare memoria, per trattare numeri casuali e per altre funzioni di utilità.
`<string.h>`	Contiene i prototipi di funzioni per le funzioni di elaborazione di stringhe.
`<time.h>`	Contiene i prototipi di funzioni e i tipi per manipolare il tempo e le date.

Fonte: Deitel & Deitel

Argomenti per valore e riferimento

- **Passaggio per Valore:**
  - Una **copia** del valore dell'argomento viene passata alla funzione chiamata
  - Le **modifiche** alla copia non influenzano il valore originale nella funzione chiamante
  - **Usato quando:** La funzione chiamata non ha bisogno di modificare il valore originale
  - **Vantaggio:** Evita effetti secondari, migliorando la correttezza e l'affidabilità del software

Argomenti per valore e riferimento

- **Passaggio per Riferimento:**
  - La funzione chiamante consente alla funzione chiamata di **modificare** il valore originale della variabile
  - **Usato quando:** La funzione chiamata deve effettivamente modificare la variabile originale
  - **Precauzione:** Utilizzare solo con funzioni affidabili per evitare modifiche indesiderate

Quiz

Generazione di numeri casuali

```
int value = rand();
```

- Genera un numero intero casuale tra 0 e `RAND_MAX`
  - costante definita nel file di intestazione `<stdlib.h>`
- Deve essere almeno 32.767 (massimo valore per un intero di due byte, cioè 16 bit).
  - 32.767 in Visual C++ di Microsoft.
  - 2.147.483.647 in gcc di GNU e Clang di Xcode
- Se la funzione è davvero casuale, ogni numero tra `0` e `RAND_MAX` ha un'uguale probabilità di essere generato

Lanciare un dado a 6 facce

```c
// fig05_04.c
// Interi casuali con spostamento e variazione di scala prodotti da 1 + rand() % 6.
#include <stdio.h>
#include <stdlib.h>

int main(void) {

   for (int i = 1; i <= 10; ++i) {
      printf("%d  ", 1 + (rand() %  6 )); // stampa valore del dado casuale
    }

    puts("");
 }
```

```
6  6  5  5  6  5  1  1  5  3
```

Fonte: Deitel & Deitel

- Utilizzando l'operatore resto effettuiamo una **variazione di scala**
- Il numero 6 è noto come **fattore di scala**
- L'output conferma che i risultati stanno nell'intervallo da 1 a 6 (l'ordine nel quale questi valori casuali sono scelti può variare in base al compilatore)

Lanciare un dado a 6 facce (60 M volte)

```c
// fig05_05.c
// Lancio di un dado a sei facce 60.000.000 di volte.
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   int frequency1 = 0; // contatore per il valore 1
   int frequency2 = 0; // contatore per il valore 2
   int frequency3 = 0; // contatore per il valore 3
   int frequency4 = 0; // contatore per il valore 4
    int frequency5 = 0; // contatore per il valore 5
    int frequency6 = 0; // contatore per il valore 6

    // ripeti 60.000.000 di volte e riepiloga i risultati
    for (int roll = 1; roll <= 60000000; ++roll) {
       int face = 1 + rand() % 6; // numero casuale da 1 a 6

       // determina il valore di face e incrementa il contatore appropriato
       switch (face) {
          case 1: // valore 1
             ++frequency1;
             break;
          case 2: // valore 2
             ++frequency2;
             break;
          case 3: // valore 3
             ++frequency3;
             break;
          case 4: // valore 4
             ++frequency4;
             break;
          case 5: // valore 5
             ++frequency5;
             break;
          case 6: // valore 6
             ++frequency6;
             break; // opzionale
       }
    }

    // stampa i risultati in formato tabellare
    printf("%s%13s\n", "Face", "Frequency");
    printf("   1%13d\n", frequency1);
    printf("   2%13d\n", frequency2);
    printf("   3%13d\n", frequency3);
    printf("   4%13d\n", frequency4);
    printf("   5%13d\n", frequency5);
    printf("   6%13d\n", frequency6);
 }
```

```
Face    Frequency
   1      9999294
   2     10002929
   3      9995360
   4     10000409
   5     10005206
   6      9996802
```

- Con gli **array** scriveremo l'equivalente dello `switch` in una singola riga!

Fonte: Deitel & Deitel

Randomizzazione del generatore di numeri casuali

- Una nuova esecuzione del programma precedente produce **sempre** la stessa sequenza di numeri
- Questa ripetibilità può sembrare contraddittoria con il concetto di "casualità", ma è una caratteristica voluta della funzione `rand`
- La ripetizione dei numeri è utile durante il **debugging**, poiché permette di testare e verificare in modo coerente le correzioni apportate al programma
- La funzione rand genera numeri **"pseudocasuali"**: una sequenza di numeri che appaiono casuali, ma che si ripete esattamente a ogni esecuzione del programma
- Questa ripetizione si verifica perché `rand` utilizza un **algoritmo deterministico** che genera la stessa sequenza a partire da un "seme" fisso

Funzione `srand`

- Per ottenere una sequenza diversa di numeri casuali a ogni esecuzione del programma, si utilizza la funzione `srand`
- `srand` accetta un argomento `int` (il "seme") e lo passa alla funzione `rand` per generare una nuova sequenza di numeri casuali
- In questo modo, ogni volta che si esegue il programma con un **seme diverso**, rand produrrà una sequenza diversa di numeri, aumentando la variabilità e simulando una **vera casualità**
- Quindi, mentre `rand` di per sé produce una sequenza deterministica di numeri pseudocasuali, `srand` consente di modificare il seme per generare sequenze diverse in ogni esecuzione del programma

Funzione `srand`

```c
// fig05_06.c
// Randomizzare il programma del lancio del dado.
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   printf("%s", "Enter seed: ");
   int seed = 0; // numero usato come seme per il generatore di numeri casuali
   scanf("%d", &seed);

   srand(seed); // fornisci il seme al generatore di numeri casuali

    for (int i = 1; i <= 10; ++i) {
       printf("%d  ", 1 + (rand() % 6)); // stampa il valore del dado casuale
    }

    puts("");
 }
```

```
Enter seed: 67
6  1  4  6  2  1  6  1  6  4


Enter seed: 867
2  4  6  1  6  1  1  3  6  2


Enter seed: 67
6  1  4  6  2  1  6  1  6  4
```

Fonte: Deitel & Deitel

Funzione `srand`

- Randomizzare il seme senza specificarlo ogni volta:

```
srand(time(NULL));
```

- `time(NULL)`: questa funzione legge l'orologio del sistema e restituisce il numero di secondi trascorsi dalla mezzanotte del 1° gennaio 1970 (notoriamente chiamata "Unix epoch")
- `srand` utilizza questo valore restituito da time come seme per il generatore di numeri casuali `rand`
- Utilizzando il valore attuale dell'orologio di sistema come seme, ogni esecuzione del programma ha un **seme diverso**, garantendo che la sequenza di numeri casuali generati da rand sia differente ogni volta
- Nota: Il **prototipo** di funzione per `time` si trova nel file di intestazione `<time.h>`

Generalizzazione dell'intervallo di numeri casuali

- I valori generati direttamente da `rand` sono sempre nell'intervallo:

  0 ≤ `rand()` ≤ **`RAND_MAX`**

- Per creare numeri casuali in qualsiasi intervallo di interi consecutivi:

```
int n = a + rand() % b;
```

- `a`: Rappresenta il **valore di spostamento**, che è il primo numero nell'intervallo desiderato
- `b`: È il **fattore di scala**, che determina l'ampiezza dell'intervallo (cioè il numero di interi consecutivi desiderati)
- Questa formula può essere adattata per generare numeri casuali in qualsiasi intervallo specificato da a (il numero iniziale) e b (l'ampiezza dell'intervallo)

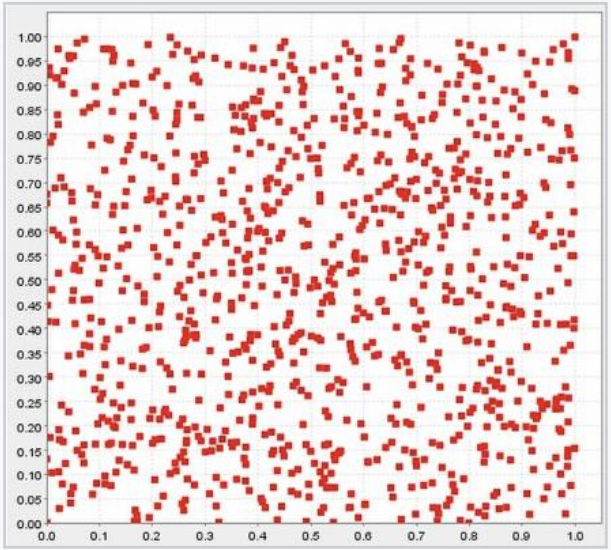
Generatore lineare congruenziale

- **Metodo classico** per generare numeri pseudo-casuali; ampiamente utilizzato grazie alla sua semplicità e velocità.
- **Formula generale**:

$$X_{n+1} = (aX_n + c) \mod m$$

  Dove:

  - X_n è il numero casuale corrente,
  - a è il moltiplicatore,
  - c è l'incremento (opzionale, nel caso di un LCG additivo),
  - m è il modulo,
  - X_0 è il **seme** o valore iniziale.
- **Periodo massimo**: un LCG può generare un massimo di m numeri distinti prima di ripetersi. Il periodo massimo si raggiunge se vengono soddisfatte certe condizioni per a, c, e m:
  - c deve essere coprimo con m,
  - $a - 1$ deve essere divisibile per tutti i fattori primi di m,
  - $a - 1$ deve essere divisibile per 4 se m è divisibile per 4.



La disposizione dei punti generati dal Generatore Lineare ran0

Fonte: Wikipedia

Intervallo

Fonte: PlaygroundAI

Caso pratico sulla simulazione di numeri casuali

- Deitel & Deitel code example (cap. 5): `fig05_07.c`
- Utilizzo di `enum`
  - un'**enumerazione** è un insieme di costanti intere rappresentate da identificatori
  - utilizzare costanti di enumerazione rende i programmi più leggibili e più facili da mantenere
- Gli identificatori all'interno di un'enumerazione devono essere **unici**
- I valori associati a questi identificatori possono essere duplicati
- Le costanti di enumerazione sono spesso scritte in lettere **maiuscole** (CONTINUE, WON, LOST) per farle risaltare nel codice e per indicare che non sono variabili modificabili

Ricorsione

- Una funzione ricorsiva **chiama sé stessa** direttamente o indirettamente attraverso un'altra funzione
- La ricorsione è particolarmente utile per risolvere problemi che possono essere suddivisi in **sottoproblemi simili** alla loro struttura originale
- La ricorsione è un argomento **complesso** e viene esplorata in dettaglio in corsi avanzati di informatica

Funzioni ricorsive

- Una funzione ricorsiva conosce solo come risolvere i casi più semplici, noti come **casi di base**
- Per i casi di base, la funzione restituisce direttamente un risultato
- Per i problemi più complessi, la funzione divide il problema in due parti:
  - Una parte che sa come risolvere
  - Una parte che non sa come risolvere
- Per rendere la ricorsione fattibile, il problema rimanente deve essere una versione più semplice o più piccola del problema originale

Chiamata ricorsiva

- La funzione chiama se stessa per risolvere il problema più semplice
- Questo passaggio è noto come chiamata **ricorsiva** o **passo di ricorsione**
- Il passo di ricorsione include un'istruzione `return` per combinare i risultati
- Il risultato della chiamata ricorsiva viene **combinato** con la porzione del problema che la funzione sa risolvere
- Questo produce un risultato che viene restituito alla funzione chiamante originale

Convergenza al caso base

- Durante il passo di ricorsione, la chiamata originaria alla funzione si arresta e attende il risultato dal passo di ricorsione
  - molte chiamate ricorsive aggiuntive mentre la funzione continua a suddividere il problema
- Per terminare la ricorsione, la sequenza di problemi più semplici deve convergere infine al **caso di base**
  - Ogni chiamata ricorsiva deve ridurre il problema verso il caso di base
- Quando la funzione riconosce il caso di base, **restituisce** un risultato alla copia precedente della funzione
- Questo avvia una sequenza di restituzioni che **risale** attraverso tutte le chiamate ricorsive
- La chiamata originaria infine restituisce il risultato finale alla sua funzione chiamante

Calcolo del fattoriale

- Il fattoriale di un intero non negativo `n` è definito come:

  $n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$

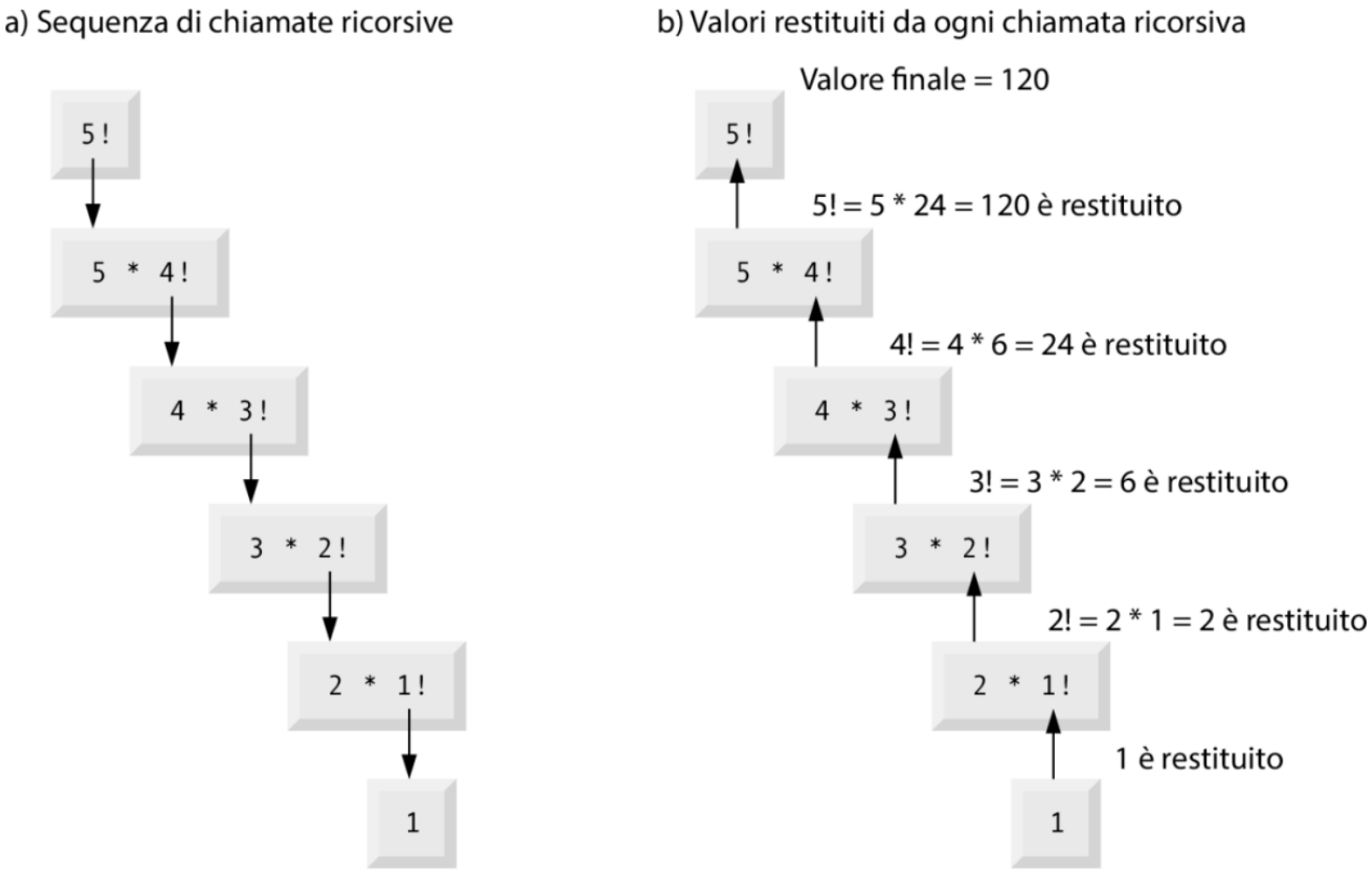
- 5! = 5 * 4 * 3 * 2 * 1 = 120
- Si può calcolare in maniera iterativa con un `for`

```
unsigned long long int factorial = 1;
for (int counter = number; counter > 1; --counter)
      factorial *= counter;
```

- Definizione ricorsiva:

  $n! = n \cdot (n - 1)!$

Calcolo del fattoriale con la ricorsione



Fonte: Deitel & Deitel

Calcolo del fattoriale con la ricorsione

```c
// fig05_09.c
// Funzione fattoriale ricorsiva.
#include <stdio.h>

unsigned long long int  factorial( int  number);

int main(void) {
   // calcolo dei fattoriali e stampa del risultato
   for (int i = 0; i <= 21; ++i) {
      printf("%d! = %llu\n", i, factorial(i) );
   }
 }

 // definizione ricorsiva della funzione fattoriale
 unsigned long long int  factorial( int  number) {
    if  (number <=  1 ) {  // caso di base
       return   1 ;
    }
    else  {  // passo ricorsivo
       return  (number * factorial(number  - 1 ));
    }
 }
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
```

Fonte: Deitel & Deitel

Funzione `factorial`

- Verifica se la condizione di terminazione è vera:
  `number <= 1`
- Altrimenti ricorre su `(number - 1)`
- L'**omissione** del caso di base o una scrittura errata del passo di ricorsione può portare a una ricorsione **infinita**
  - Questo porta all'esaurimento della memoria
  - È simile a un ciclo infinito in una soluzione iterativa, ma i cicli infiniti di solito non esauriscono la memoria

Crescita rapida dei fattoriali

- La funzione `factorial` restituisce un risultato di tipo `unsigned long long int`
  - Questo tipo può rappresentare valori fino a 18.446.744.073.709.551.615
- I fattoriali crescono rapidamente e superano facilmente il valore massimo di `unsigned long long int`
- Si possono calcolare fattoriali solo fino a circa 21!
- Linguaggi come il C non possono essere facilmente estesi per gestire numeri molto grandi
  - Linguaggi a oggetti superano questa limitazione con le classi

La serie di Fibonacci

- Inizia con 0 e 1
- Ogni numero successivo è la somma dei due numeri precedenti.
- Esempio: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- La serie di Fibonacci appare frequentemente in natura e descrive forme a spirale
- Il rapporto tra numeri successivi converge verso 1,618..., noto come **proporzione aurea** o **sezione aurea**
- Gli esseri umani tendono a trovare la sezione aurea esteticamente attraente
- Architetti e designer usano spesso la proporzione aurea per progettare finestre, stanze e edifici

La serie di Fibonacci con la ricorsione

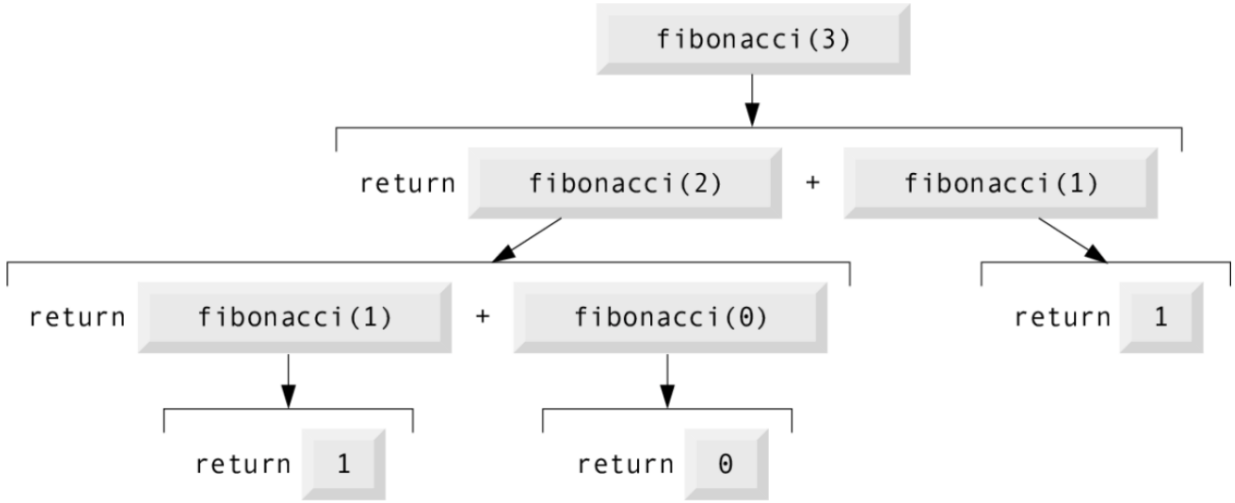
```c
// fig05_10.c
// Funzione ricorsiva fibonacci.
#include <stdio.h>

unsigned long long int  fibonacci( int  n); // prototipo di funzione

int main(void) {
   // calcola e stampa fibonacci(number) per 0-10
   for (int number = 0; number <= 10; number++) {
      printf("Fibonacci(%d) = %llu\n", number, fibonacci(number) );
   }

    printf("Fibonacci(20) = %llu\n", fibonacci( 20 ) );
    printf("Fibonacci(30) = %llu\n", fibonacci( 30 ) );
    printf("Fibonacci(40) = %llu\n", fibonacci( 40 ) );
}

// Definizione ricorsiva della funzione fibonacci
unsigned long long int  fibonacci(int n) {
   if ( 0  == n ||  1  == n) {  // caso di base
       return  n;
   }
    else  {  // passo ricorsivo
       return  fibonacci(n -  1 ) + fibonacci(n -  2 );
   }
}
```

```
Fibonacci(0) = 0
Fibonacci(1) = 1
Fibonacci(2) = 1
Fibonacci(3) = 2
Fibonacci(4) = 3
Fibonacci(5) = 5
Fibonacci(6) = 8
Fibonacci(7) = 13
Fibonacci(8) = 21
Fibonacci(9) = 34
Fibonacci(10) = 55
Fibonacci(20) = 6765
Fibonacci(30) = 832040
Fibonacci(40) = 102334155
```

Fonte: Deitel & Deitel

La serie di Fibonacci con la ricorsione

- È interessante notare che, se `n` è maggiore di 1, il passo di ricorsione genera *due* chiamate ricorsive, ognuna delle quali risolve un problema leggermente più semplice dell'originaria chiamata a `fibonacci`



Fonte: Deitel & Deitel

Ordine di valutazione degli operandi

- Il C non specifica l'ordine in cui gli operandi degli operatori sono **valutati**
- Durante la valutazione di `fibonacci(3)`, vengono effettuate chiamate ricorsive a `fibonacci(2)` e `fibonacci(1)`
  - L'ordine in cui queste chiamate ricorsive vengono eseguite non è definito dal C
- Sebbene l'ordine di applicazione degli operatori sia determinato dalle regole di precedenza e associatività, l'ordine di valutazione degli operandi può variare
- In molti casi, il risultato finale rimane lo stesso indipendentemente dall'ordine di valutazione.
- In programmi complessi, la valutazione non deterministica degli operandi può avere effetti secondari che influenzano il risultato finale

Ordine di valutazione degli operandi

- Il C specifica l'ordine di valutazione degli operandi per quattro operatori: &&, ||, , (operatore virgola) e ?: (operatore ternario)
- Per gli operatori **binari** && (AND logico), || (OR logico) e , (virgola), gli operandi vengono valutati da sinistra a destra
  - Nota: Le virgole usate per separare gli argomenti nelle chiamate di funzione non sono operatori virgola e non seguono questa regola
- Operatore **ternario** ?:
  - L'ordine di valutazione è specifico: l'operando più a sinistra (la condizione) viene valutato per primo
  - Se l'operando più a sinistra è diverso da zero (vero), viene valutato l'operando centrale e l'ultimo operando viene ignorato
  - Se l'operando più a sinistra è uguale a zero (falso), viene valutato il terzo operando e l'operando centrale viene ignorato

Complessità esponenziale

- Nella funzione di Fibonacci ogni chiamata ricorsiva raddoppia il numero di chiamate
- Per calcolare l'n^{mo} numero serviranno un ordine 2^n chiamate
  - 20esimo numero = 2^{20} ~ 1 M di chiamate
- Questo fenomeno si chiama **complessità esponenziale** in Informatica
- Sebbene la soluzione ricorsiva sia intuitiva, esistono approcci migliori per calcolare i numeri di Fibonacci che possono essere più **efficienti**
- I problemi con complessità esponenziale sono studiati in dettaglio nei corsi avanzati di informatica, specialmente quelli riguardanti gli algoritmi

Ricorsione o iterazione

- Strutture di controllo:
  - Selezione **vs** iterazione
- Ripetizione:
  - chiamata a funzione **vs** iterazione
- Terminazione:
  - caso di base **vs** condizione del ciclo
- Procedimento:
  - versioni più semplici del problema **vs** modifica contatore o variabile

Errori comuni

- Iterazione infinita:
  - spesso dovuta a errori nella logica che impediscono al ciclo di terminare
- Ricorsione infinita:
  - spesso dovuta a errori nella logica che impediscono al problema di ridursi al caso di base

Overhead ricorsione

- La ricorsione invoca ripetutamente il meccanismo di chiamata di funzione, creando un aggravio di calcolo e di spazio di memoria
- Ogni chiamata ricorsiva crea una nuova copia della funzione (solo variabili, non l'intero codice), il che può consumare molta memoria

La ricorsione non è richiesta

- Qualunque problema risolvibile in modo ricorsivo può essere risolto anche in modo iterativo
- Preferita quando il problema si adatta naturalmente a una soluzione ricorsiva e produce un codice più leggibile e manutenibile
- In alcuni casi, la soluzione iterativa può non essere immediatamente evidente o può essere più complessa da implementare rispetto a una soluzione ricorsiva

Esempio: da ricorsione ad iterazione

Versione ricorsiva (Potenza)

c — Copia codice

```c
#include <stdio.h>

// Funzione ricorsiva per calcolare x^n
int potenza(int base, int esponente) {
    if (esponente == 0) {
        return 1;
    } else {
        return base * potenza(base, esponente - 1);  // Chiamata ricorsiva
    }
}

int main() {
    int base = 2, esponente = 5;
    printf("%d^%d = %d\n", base, esponente, potenza(base, esponente));
    return 0;
}
```

Versione iterativa (Potenza)

c — Copia codice

```c
#include <stdio.h>

// Funzione iterativa per calcolare x^n
int potenza(int base, int esponente) {
    int risultato = 1;
    for (int i = 0; i < esponente; i++) {
        risultato *= base;
    }
    return risultato;
}

int main() {
    int base = 2, esponente = 5;
    printf("%d^%d = %d\n", base, esponente, potenza(base, esponente));
    return 0;
}
```

Fonte: chatGPT

Recap

- Modularizzazione dei programmi con le funzioni
- Definizione funzioni e prototipi
- Pila delle chiamate di funzione e record di attivazione
- Passare argomenti
- Ricorsione