Fondamenti di Programmazione (A)

17 - Funzioni (visibilità, call stack e funzioni ricorsive)

Vincenzo Arceri - Università degli Studi di Parma - vincenzo.arceri@unipr.it

Puntate precedenti

- Funzioni
- Passaggio di parametri
  - Passaggio per valore
  - Passaggio per riferimento
  - Differenze fra C e C++

Scoping

- Il **campo d'azione (*scope*)** di una dichiarazione è l'insieme di parti di programma in cui il binding nome-oggetto denotabile è **visibile**

Scoping

- Il **campo d'azione (*scope*)** di una dichiarazione è l'insieme di parti di programma in cui il binding nome-oggetto denotabile è **visibile**

- Regole di scope: sono le regole che determinano la visibilità di un'associazione nome-oggetto denotabile

Scoping
Funzioni

```
int g, h;

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
}
```

x: variabile locale a `main`

Scoping
Funzioni

```
int g, h;

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
}
```

`result`: variabile locale a `fact`

Scoping
Funzioni

```
int g, h;

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
}
```

`i`: variabile locale a `fact`

Scoping
Funzioni

```
int g, h;

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
}
```

`fact`: visibile dalla sua dichiarazione

Scoping
Funzioni

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

Scoping

Funzioni

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

Scoping
Funzioni

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

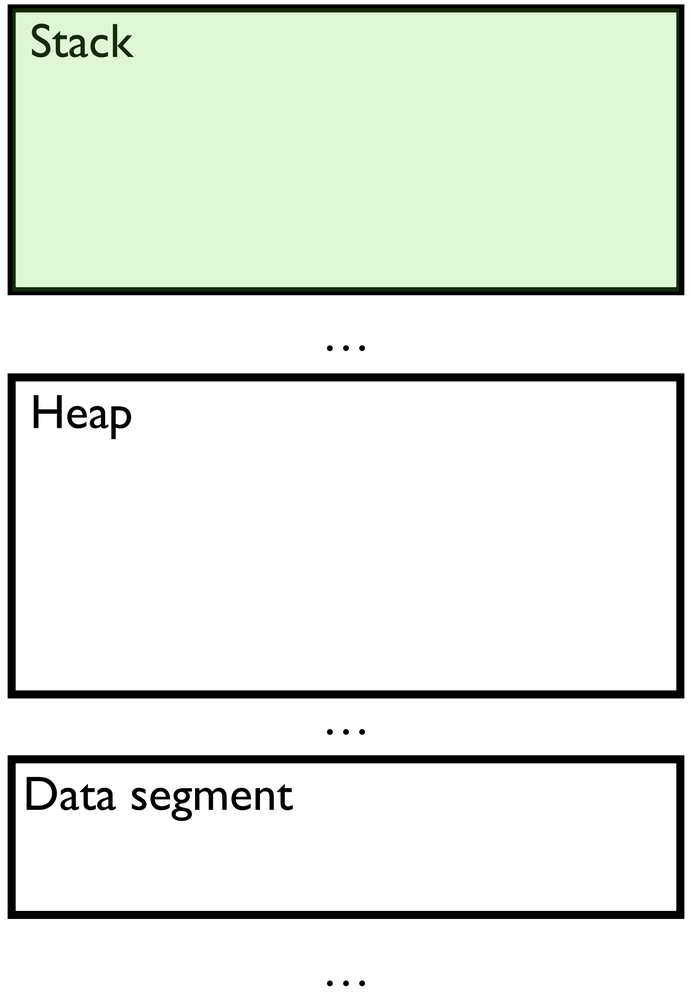
Variabili locali: esistono in memoria solo quando la funzione che la contiene è attiva

Layout della memoria

C++

Layout della memoria

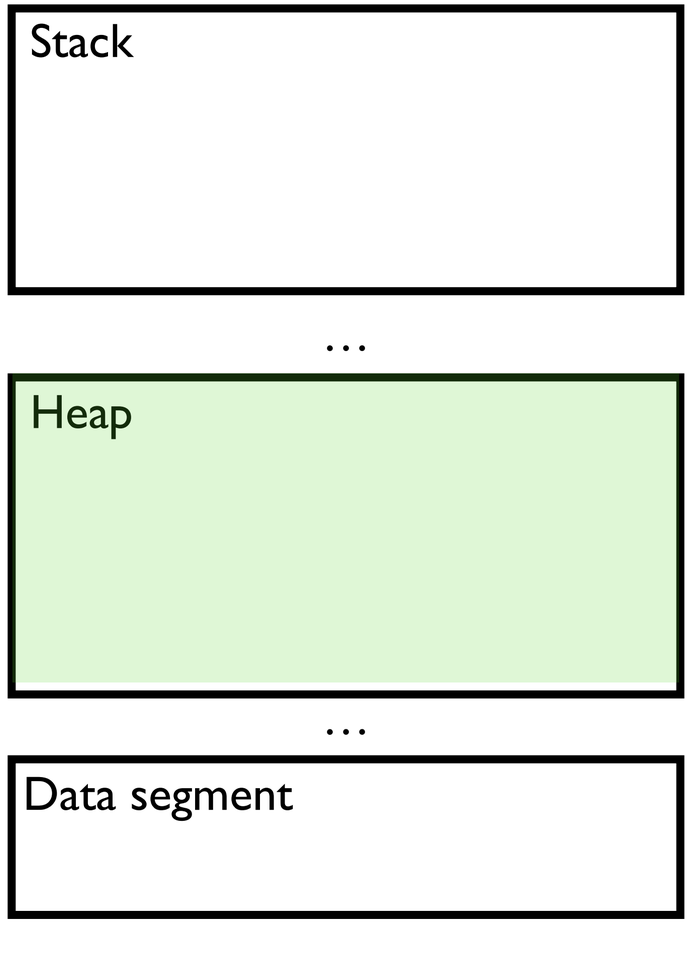
C++



Variabili locali, passare argomenti alle funzioni, ritornare valori alle funzioni, gestire le chiamate a funzioni

Layout della memoria

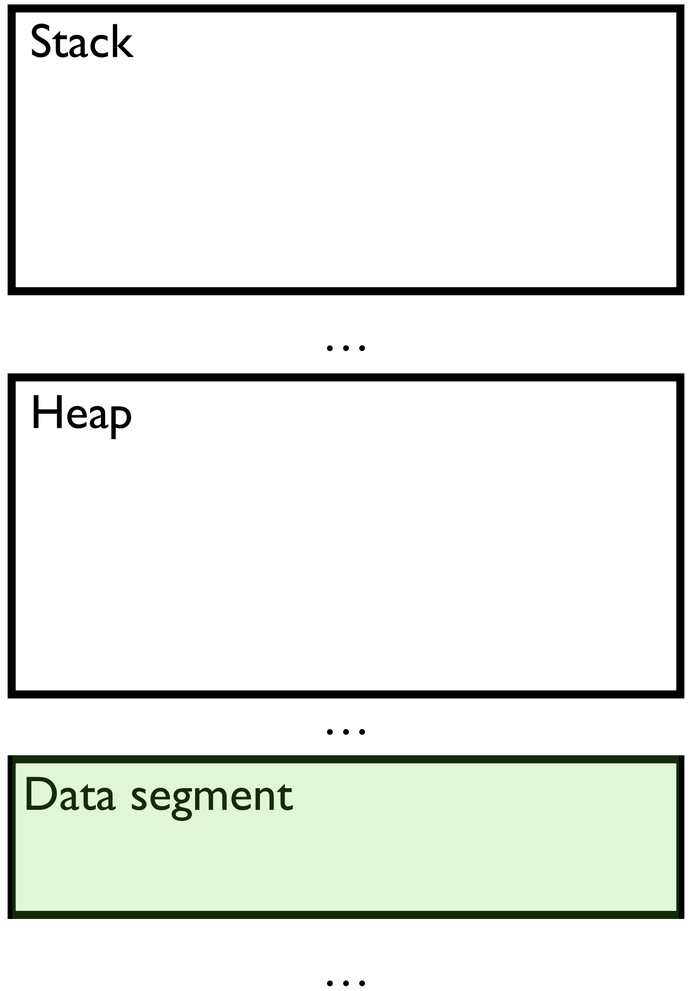
C++



Allocazione dinamica della memoria (prossime lezioni)

Layout della memoria

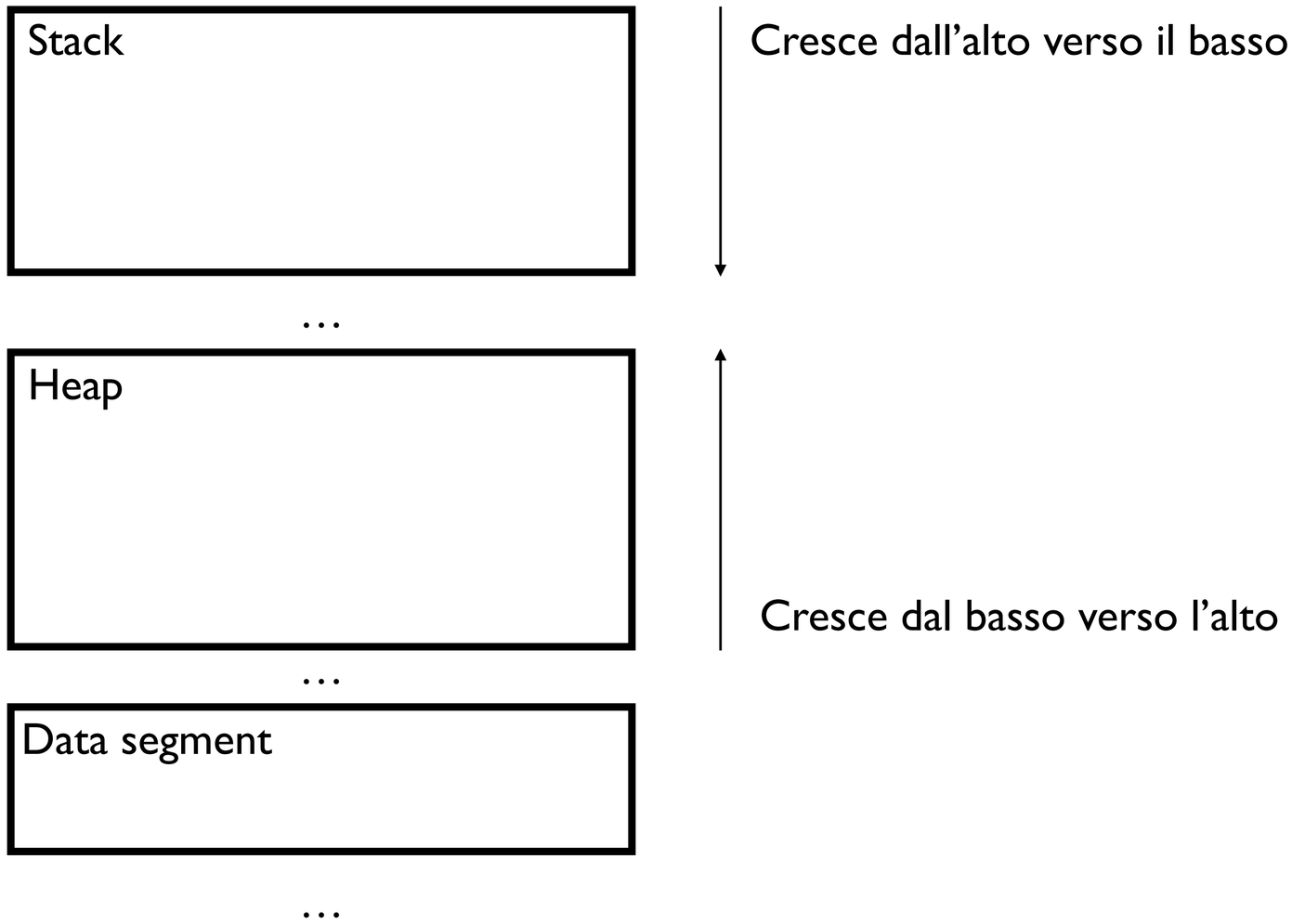
C++



Variabili globali e statiche

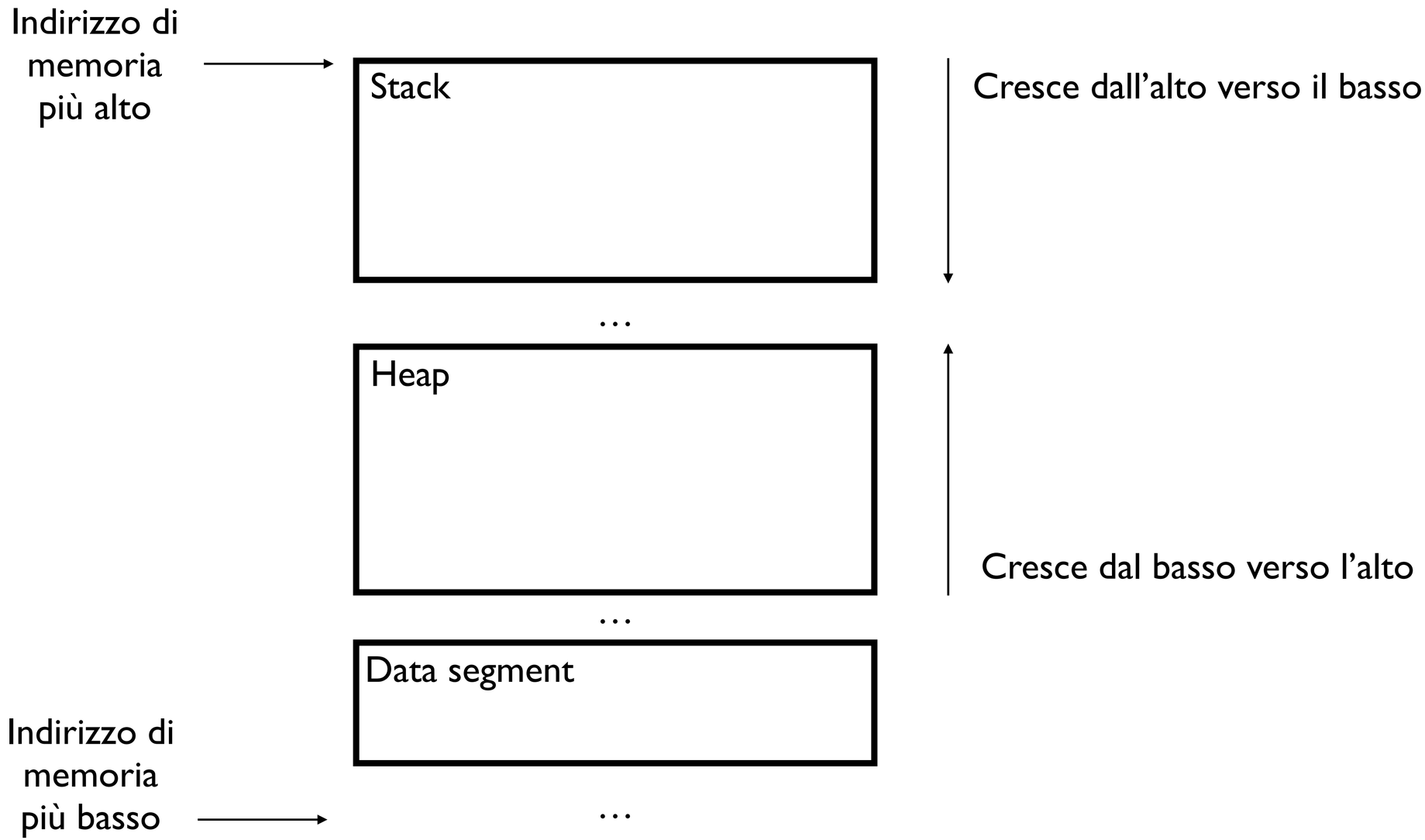
Layout della memoria

C++



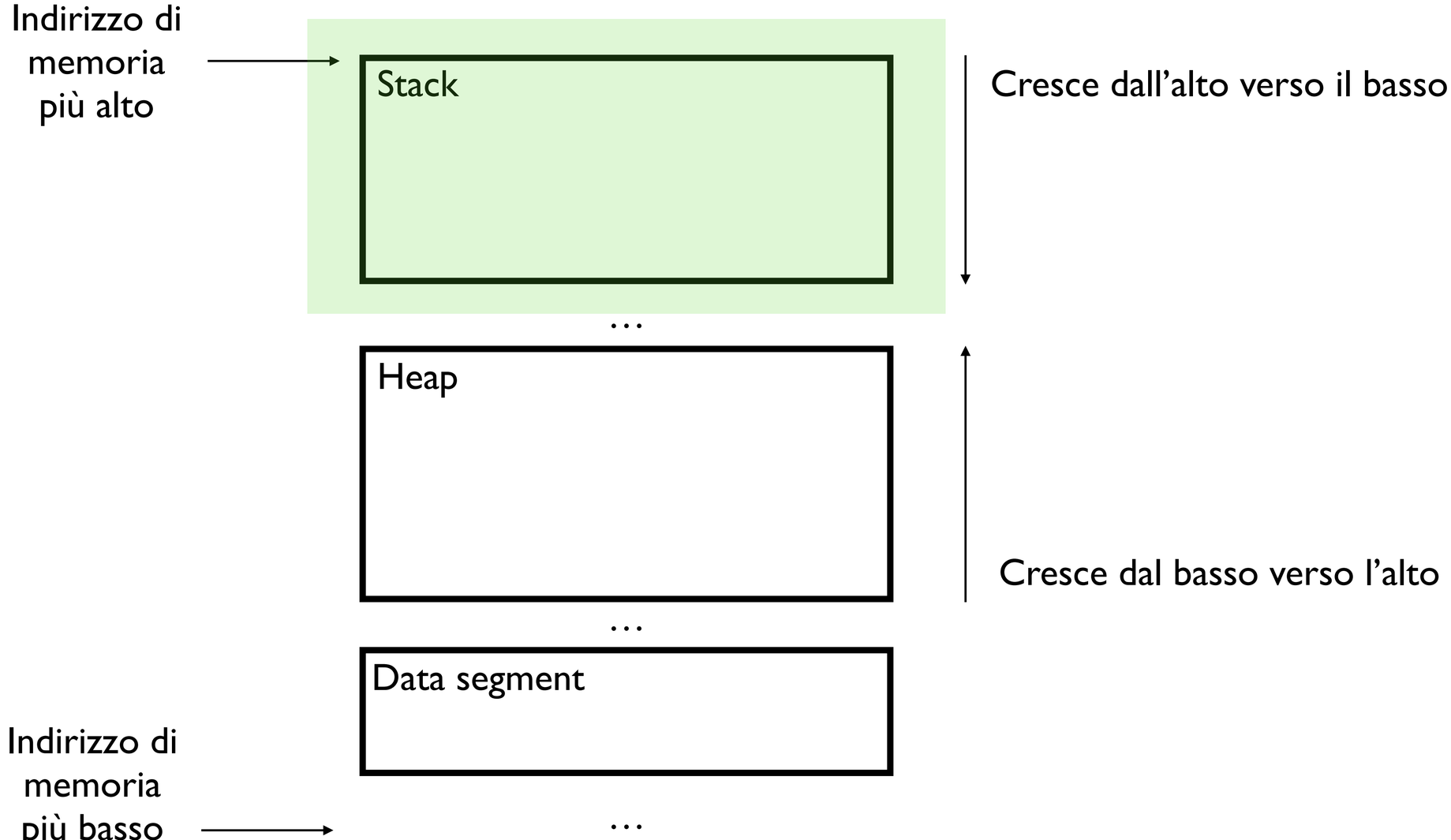
Layout della memoria

C++



Layout della memoria

C++



Stack

- Struttura di dati astratta con accesso LIFO (*Last In First Out*)
- Call stack, execution stack, function stack,…
- Zona di memoria del programma utilizzata per gestire le **chiamate** di funzione **attive**
- Cresce dall'alto verso il basso
- Variabili locali, parametri, valori di ritorno, …
- Versione semplificata: più dettagli al corso di Sistemi Operativi (2° anno)

Stack

LIFO

Stack

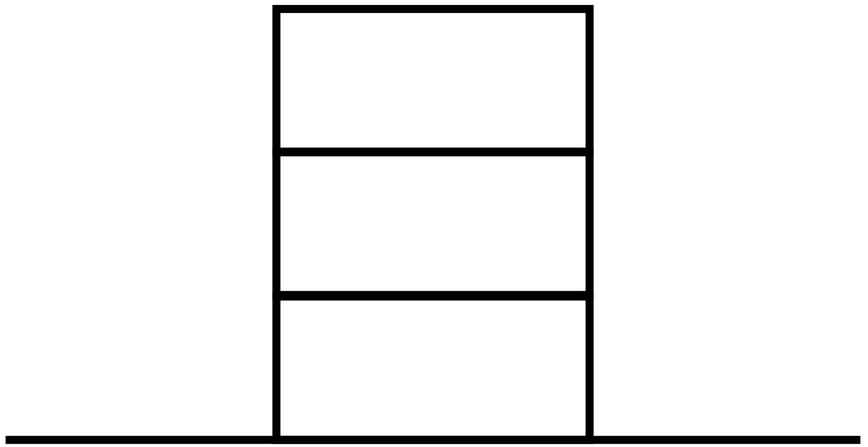
LIFO

Stack

LIFO

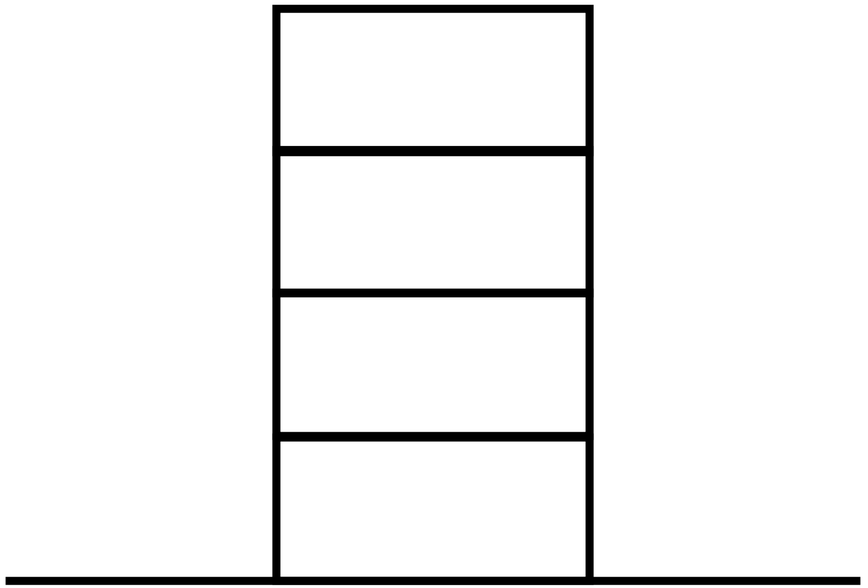
Stack

LIFO



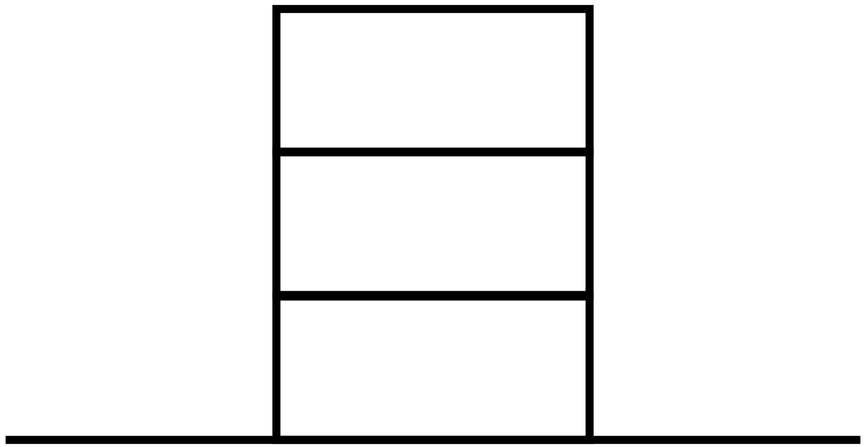
Stack

LIFO



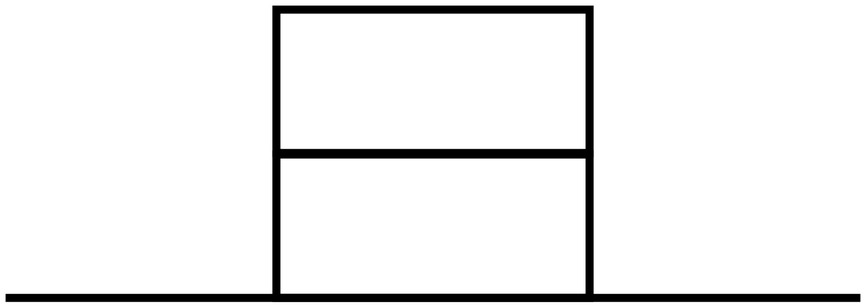
Stack

LIFO

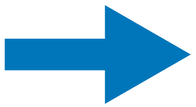


Stack

LIFO



Stack



```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

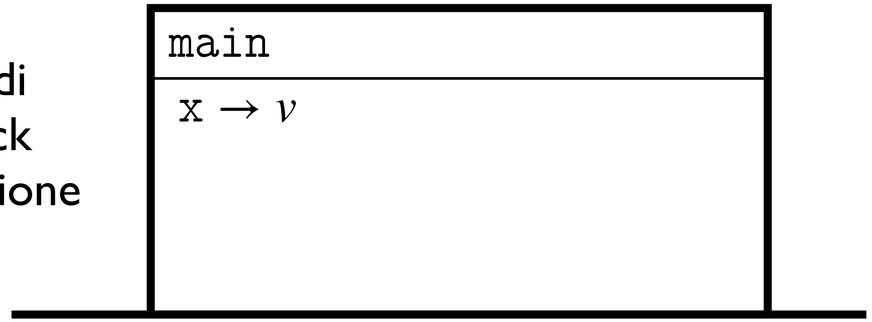
Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

Stack frame: zona di memoria dello stack riservato ad una funzione attiva

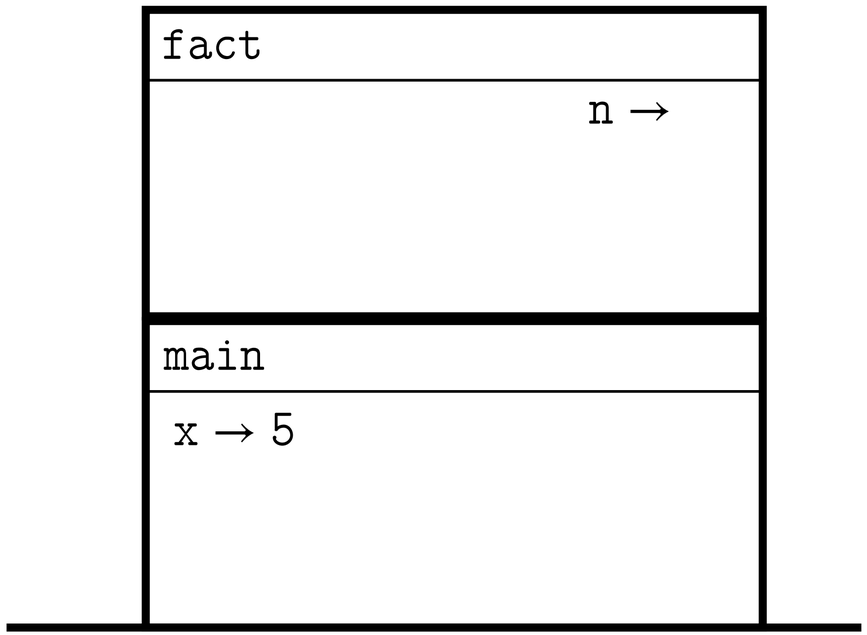


Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```



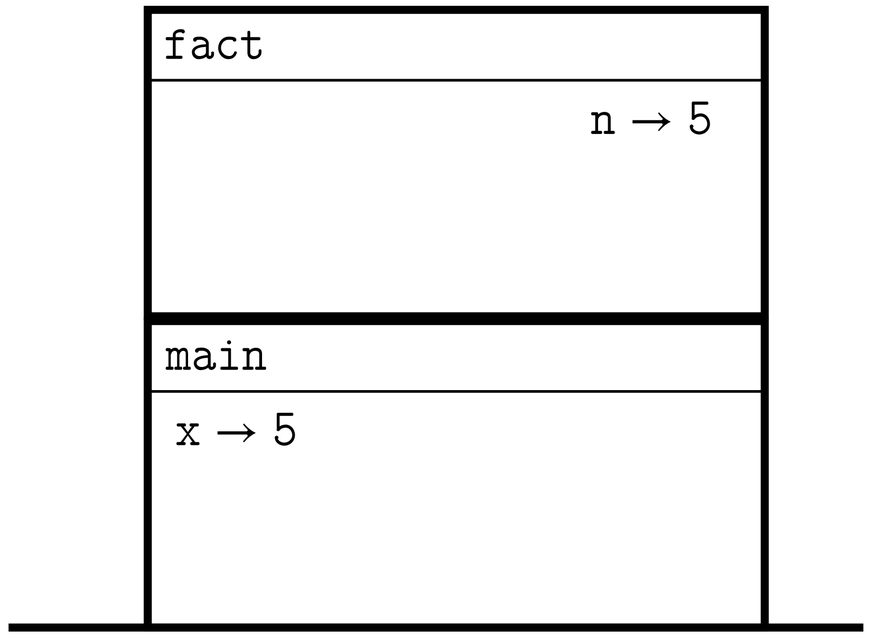
Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

Passaggio dei parametri



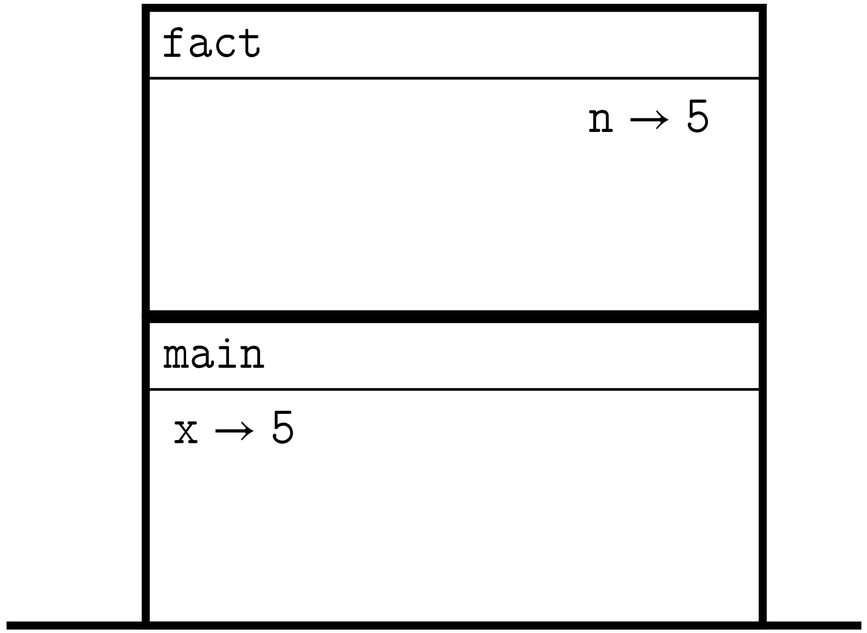
Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

Il controllo passa alla funzione `fact`

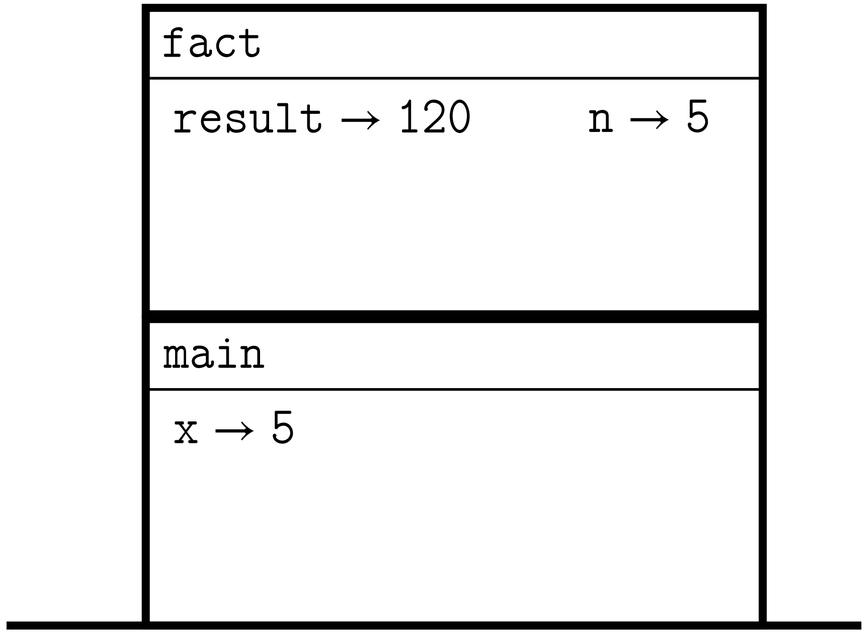


Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```



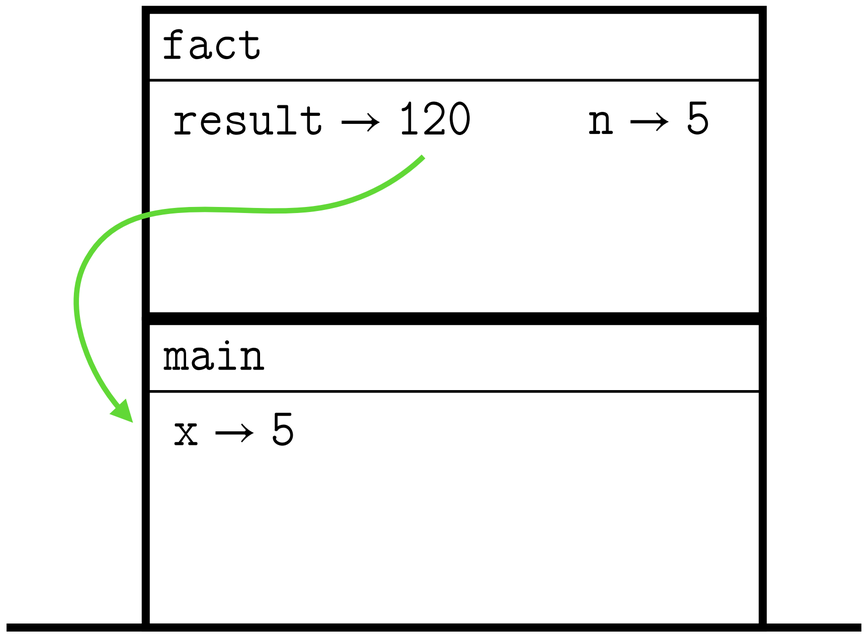
Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

La funzione `fact` ritorna al chiamante



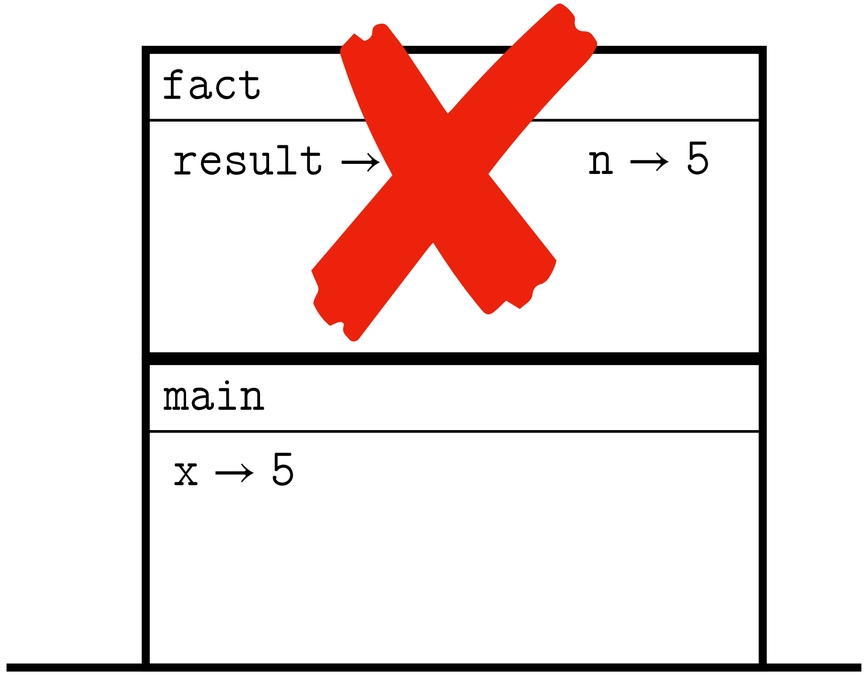
Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

Il controllo passa nuovamente alla funzione `main`

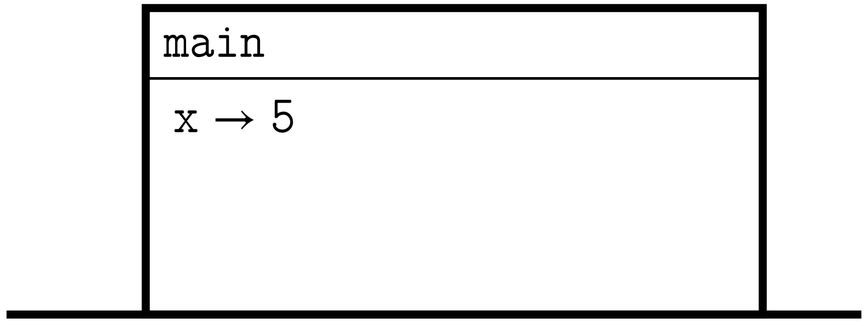


Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

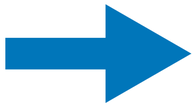


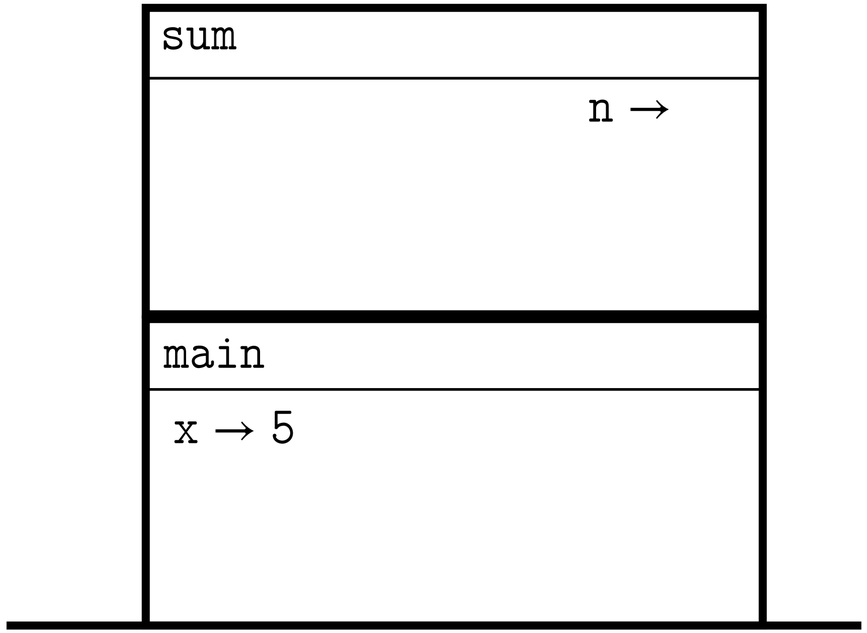
Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```



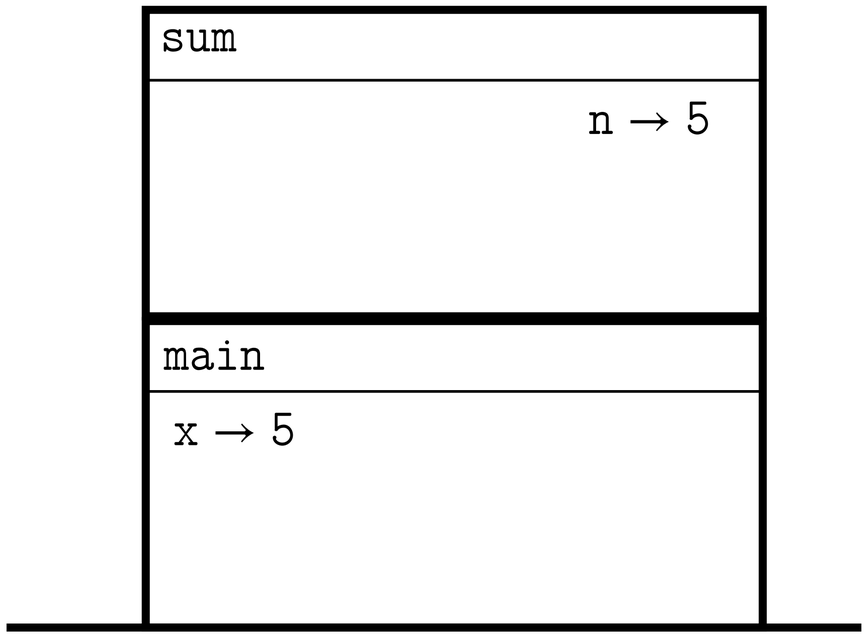


Stack

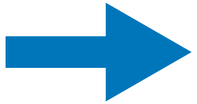
```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```



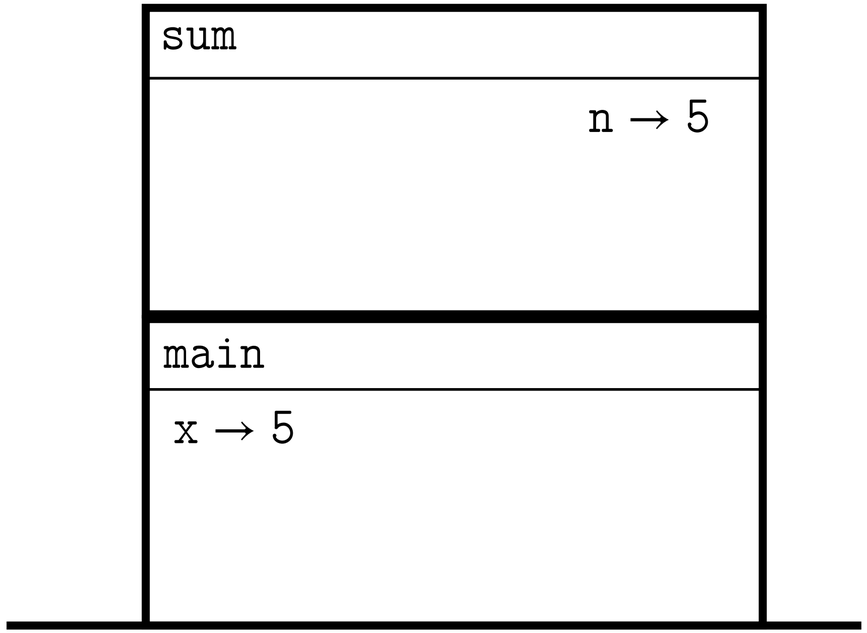
Stack



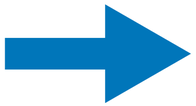
```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```



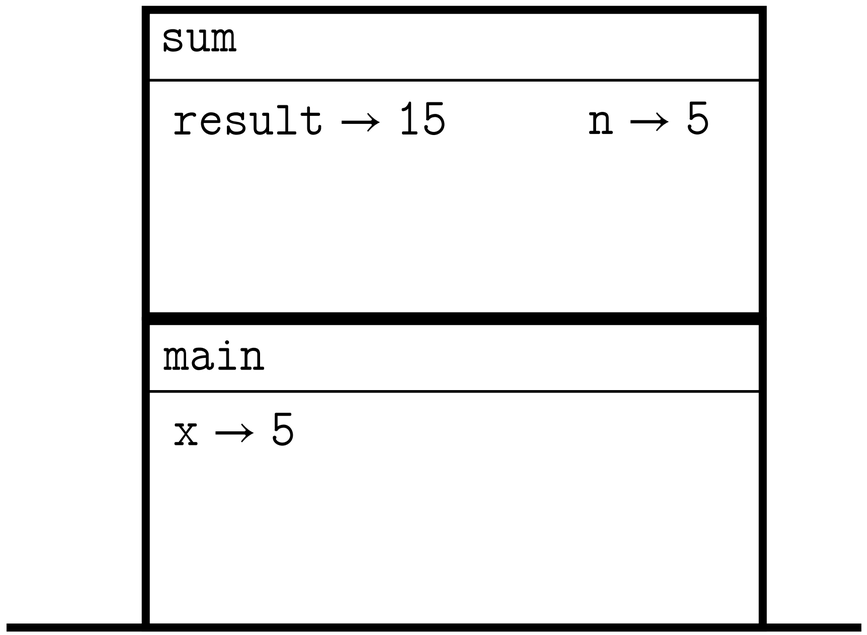
Stack



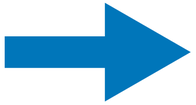
```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```



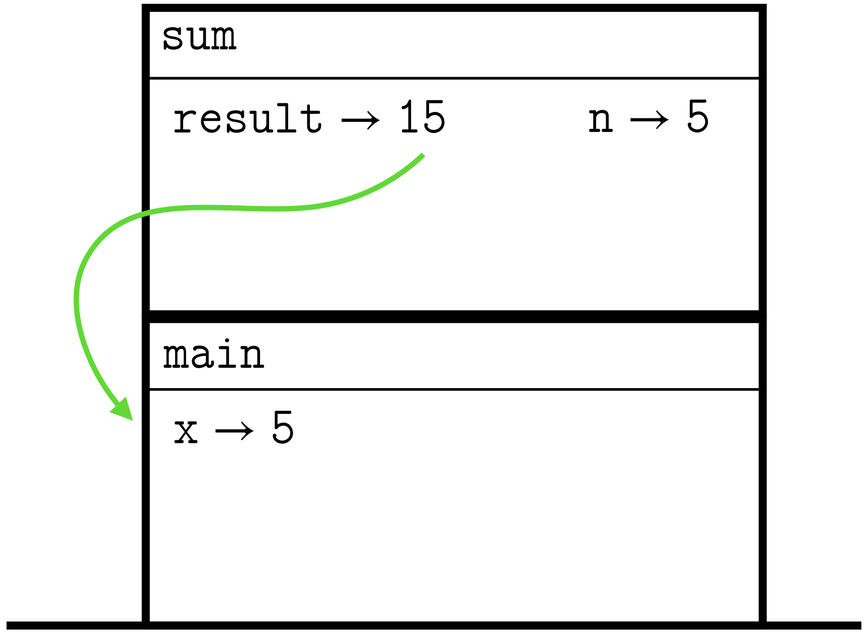
Stack



```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

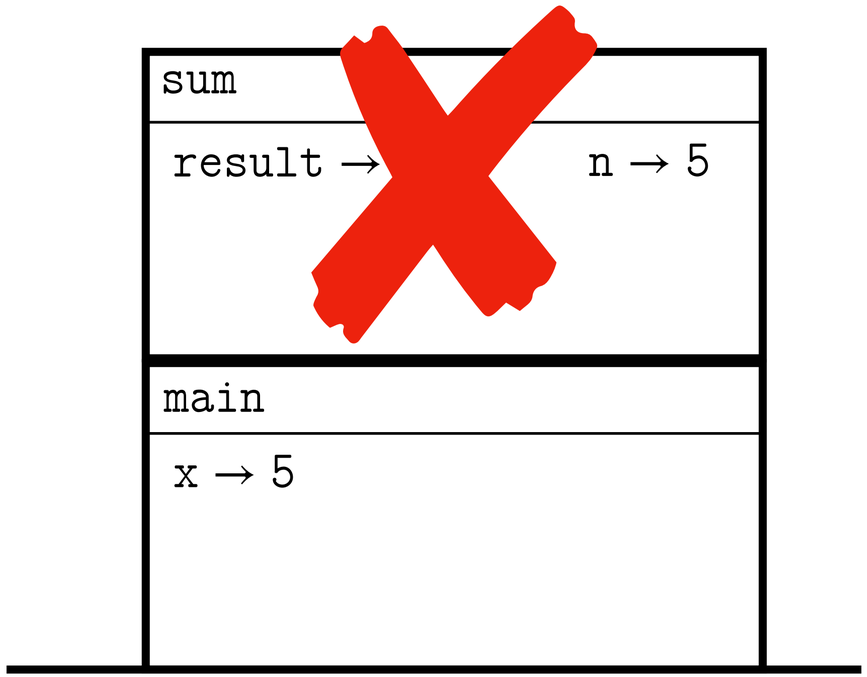


Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

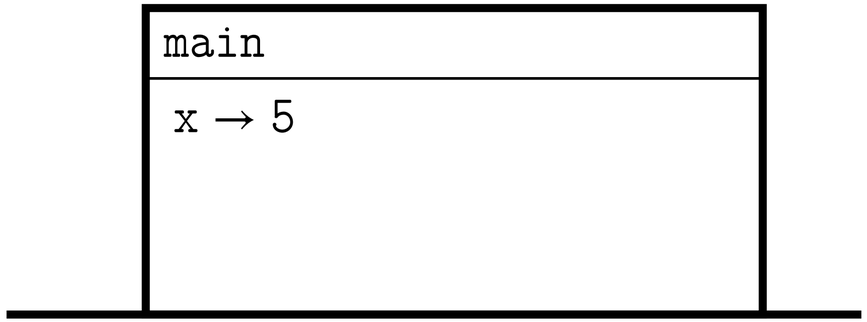


Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

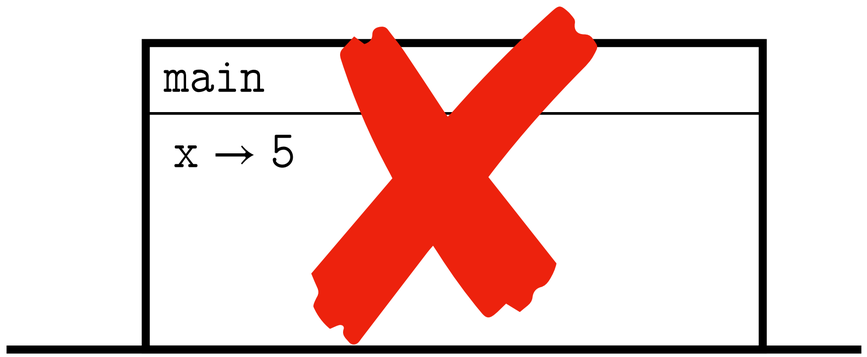


Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```

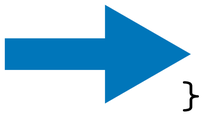


Stack

```
int sum(int n) {
    int result = 0;
    for (int i = 0; i <= n; i++)
        result += i;
    return result;
}

int fact(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
        result *= i;
    return result;
}

int main() {
  int x;
  cin >> x;
  cout << fact(x) << endl;
  cout << sum(x) << endl;
  return 0;
}
```



Fattoriale

$$fact : \mathbb{N} \to \mathbb{N}$$

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

Fattoriale

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(7) = 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5040$$

Fattoriale

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(7) = 7 \cdot \underbrace{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(6)} = 5040$$

Fattoriale

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(7) = 7 \cdot \underbrace{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(6)} = 5040$$

$$fact(6) = 6 \cdot \underbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(5)} = 720$$

Fattoriale

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(7) = 7 \cdot \underbrace{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(6)} = 5040$$

$$fact(6) = 6 \cdot \underbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(5)} = 720$$

$$fact(5) = 5 \cdot \underbrace{4 \cdot 3 \cdot 2 \cdot 1}_{fact(4)} = 120$$

Fattoriale

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(7) = 7 \cdot \underbrace{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(6)} = 5040$$

$$fact(6) = 6 \cdot \underbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(5)} = 720$$

$$fact(5) = 5 \cdot \underbrace{4 \cdot 3 \cdot 2 \cdot 1}_{fact(4)} = 120$$

$$fact(4) = 4 \cdot \underbrace{3 \cdot 2 \cdot 1}_{fact(3)} = 24$$

Fattoriale

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(7) = 7 \cdot \underbrace{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(6)} = 5040$$

$$fact(6) = 6 \cdot \underbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(5)} = 720$$

$$fact(5) = 5 \cdot \underbrace{4 \cdot 3 \cdot 2 \cdot 1}_{fact(4)} = 120$$

$$fact(4) = 4 \cdot \underbrace{3 \cdot 2 \cdot 1}_{fact(3)} = 24$$

$$fact(3) = 3 \cdot \underbrace{2 \cdot 1}_{fact(2)} = 6$$

Fattoriale

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(7) = 7 \cdot \underbrace{6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(6)} = 5040$$

$$fact(6) = 6 \cdot \underbrace{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}_{fact(5)} = 720$$

$$fact(5) = 5 \cdot \underbrace{4 \cdot 3 \cdot 2 \cdot 1}_{fact(4)} = 120$$

$$fact(4) = 4 \cdot \underbrace{3 \cdot 2 \cdot 1}_{fact(3)} = 24$$

$$fact(3) = 3 \cdot \underbrace{2 \cdot 1}_{fact(2)} = 6$$

$$fact(2) = 2 \cdot \underbrace{1}_{fact(1) = 1} = 2$$

Funzioni ricorsive

$$fact : \mathbb{N} \to \mathbb{N}$$

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot fact(n-1) & \text{otherwise} \end{cases}$$

Funzioni ricorsive

Funzioni ricorsive

- Una funzione è detta ricorsiva se la sua definizione è espressa in termini di se stessa

Funzioni ricorsive

- Una funzione è detta ricorsiva se la sua definizione è espressa in termini di se stessa
- Una funzione ricorsiva richiama se stessa su un'istanza dell'input *semplificata*

Funzioni ricorsive

- Una funzione è detta ricorsiva se la sua definizione è espressa in termini di se stessa
- Una funzione ricorsiva richiama se stessa su un'istanza dell'input *semplificata*

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot fact(n-1) & \text{otherwise} \end{cases}$$

Funzioni ricorsive

- Una funzione è detta ricorsiva se la sua definizione è espressa in termini di se stessa
- Una funzione ricorsiva richiama se stessa su un'istanza dell'input *semplificata*

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot fact(n-1) & \text{otherwise} \end{cases}$$

- Una funzione ricorsiva deve avere:
  - Una (o più) chiamate ricorsive
  - Uno (o più) casi base

Funzioni ricorsive

Iterazione vs ricorsione

Funzioni ricorsive

Iterazione vs ricorsione

- Qualsiasi funzione ricorsiva può essere riscritta come una funzione iterativa e viceversa

Funzioni ricorsive

Iterazione vs ricorsione

- Qualsiasi funzione ricorsiva può essere riscritta come una funzione iterativa e viceversa

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot fact(n-1) & \text{otherwise} \end{cases}$$

Funzioni ricorsive

Iterazione vs ricorsione

- Qualsiasi funzione ricorsiva può essere riscritta come una funzione iterativa e viceversa

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot (n-1) \cdot \ldots \cdot 1 & \text{otherwise} \end{cases}$$

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \vee n = 1 \\ n \cdot fact(n-1) & \text{otherwise} \end{cases}$$

- Funzioni ricorsive: soluzioni più chiare ed eleganti
- Funzioni iterative: soluzioni più efficienti

Esercizio

- Problema: si scriva una funzione ricorsiva che prende in input due interi x e y e calcoli x^y

Esercizio

- Problema: si scriva una funzione ricorsiva che prende in input un intero i e calcoli l'i-esimo numero di Fibonacci

Esercizio

- Problema: si scriva una funzione ricorsiva che presa in input una stringa, calcoli la sua lunghezza

Esercizio

- Problema: si scriva una funzione ricorsiva che prende in input un array di interi e ritorna il massimo valore dell'array

Esercizio

- Problema: si scriva una funzione ricorsiva che prende in input un array di interi e un intero x e ritorna `true` se l'array contiene x, `false` altrimenti