



SAPIENZA
UNIVERSITÀ DI ROMA

Algoritmi

Volume 2 — Linguaggi di programmazione

Dallo pseudocodice ai programmi: tre linguaggi di programmazione, C++ (compilato), Python (interpretato) e Java (compilato in bytecode ed eseguito da una macchina virtuale). Per ciascuno: variabili e tipi di dato, input e output, operatori, istruzioni condizionali, cicli, array e liste, funzioni, operatori bit a bit e un algoritmo completo, con il confronto fra i tre linguaggi.

Fabio Furini

fabiofurini.github.io

Dipartimento di Ingegneria informatica, automatica e gestionale — Sapienza Università di Roma

Versione online e materiale scaricabile:

fabiofurini.github.io/algoritmi/
github.com/fabiofurini/algoritmi

Anno accademico 2026–2027

Indice

6	Introduzione a C++	3
	Introduzione	3
6.1	Caratteristiche principali del linguaggio di programmazione	4
6.2	Variabili e tipi di dato	5
6.3	Flusso di ingresso e uscita	6
6.4	Operatori	7
6.5	Istruzioni condizionali	12
6.6	Cicli	14
6.7	Array	16
6.8	La funzione <code>main</code> e le funzioni definite dall'utente	19
6.9	Un algoritmo completo: il valore massimo in un array	21
6.10	Operatori bit a bit	23
7	Introduzione a Python	31
	Introduzione	31
7.1	Caratteristiche principali del linguaggio di programmazione	32
7.2	Variabili e tipi di dato	34
7.3	Ingresso e uscita	35
7.4	Operatori	36
7.5	Istruzioni condizionali	42
7.6	Cicli	44
7.7	Liste	46
7.8	Funzioni	49
7.9	Un algoritmo completo: il valore massimo in un array	52
7.10	Operatori bit a bit	54
7.11	Python e C++ a confronto	55
8	Introduzione a Java	56
	Introduzione	56
8.1	Caratteristiche principali del linguaggio di programmazione	57
8.2	Variabili e tipi di dato	58
8.3	Ingresso e uscita	60
8.4	Operatori	61

8.5	Istruzioni condizionali	66
8.6	Cicli	68
8.7	Array	69
8.8	Metodi (funzioni)	72
8.9	Un algoritmo completo: il valore massimo in un array	75
8.10	Operatori bit a bit	78
8.11	Java, C++ e Python a confronto	79

Introduzione a C++

Introduzione

Nel capitolo «Pseudocodice» gli algoritmi sono scritti in un linguaggio pensato per le persone: è preciso, ma non viene eseguito da un calcolatore. Per eseguire un algoritmo dobbiamo tradurlo in un **linguaggio di programmazione**. In questo capitolo introduciamo il linguaggio di programmazione C++, che usiamo per implementare gli algoritmi del corso.

Il capitolo segue lo stesso percorso del capitolo «Pseudocodice»:

- le caratteristiche principali del linguaggio e la struttura di un programma;
- variabili, tipi di dato e flusso di ingresso e uscita;
- operatori, istruzioni condizionali e cicli;
- gli array, la cui dimensione può essere fissata o decisa durante l'esecuzione;
- le funzioni, con la funzione `main` come punto di ingresso di ogni programma;
- un algoritmo completo, la ricerca del valore massimo in un array, scritto in pseudocodice e in C++;
- gli operatori bit a bit, che agiscono direttamente sulla rappresentazione binaria dei numeri interi.

Ogni frammento di codice del capitolo è stato compilato con il compilatore g++ (standard C++17) ed eseguito: gli output riportati negli esempi sono quelli stampati dai programmi.

Nello pseudocodice le componenti di un array $a[1..n]$ sono indicizzate da 1 a n ; in C++ le componenti di un array di dimensione n sono indicizzate da 0 a $n - 1$. La componente $a[i]$ dello pseudocodice corrisponde alla componente `a[i-1]` del codice C++.

6.1 Caratteristiche principali del linguaggio di programmazione

Le caratteristiche principali di C++ sono le seguenti.

- C++ è un linguaggio **imperativo** che dà accesso diretto alla memoria e alle risorse hardware del calcolatore. È molto efficiente per la programmazione di sistema a basso livello e allo stesso tempo supporta astrazioni di alto livello, il che lo rende versatile sia per le applicazioni in cui le prestazioni sono critiche sia per lo sviluppo software moderno.
- C++ supporta le **funzioni**, che permettono un approccio strutturato e modulare alla programmazione. Le funzioni consentono di riutilizzare il codice e di organizzarlo meglio.
- Il linguaggio di programmazione C++ comprende un insieme predefinito di **simboli riservati** usati per dichiarare variabili, definire funzioni e specificare le istruzioni eseguibili. Il **codice sorgente** è costituito da una sequenza di **operazioni atomiche** eseguite in ordine, una dopo l'altra, su variabili di **tipi di dato** specifici.
- Il codice sorgente in C++ può contenere **commenti**, che servono come documentazione del codice e vengono ignorati dal compilatore. In C++, un commento su una sola riga inizia con il simbolo riservato `//`. I commenti su più righe sono racchiusi fra i simboli riservati `/*` e `*/`.

Esempio 6.1: commenti in C++

Questo è un esempio di commento su una sola riga:

```
1 // Questa è una riga di commento
```

Questo è un esempio di commento su più righe:

```
1 /* This is a  
2    multi-line comment */
```

6.1.1 La struttura di un programma

Prima di descrivere le singole istruzioni, osserviamo lo scheletro che ogni programma ha in comune. Un programma C++ è un file di testo (con estensione `.cpp`) che viene tradotto da un **compilatore** in un file eseguibile. L'esecuzione del programma parte dalla funzione `main`, il punto di ingresso di ogni programma.

Esempio 6.2: lo scheletro di un programma

```
1 #include <iostream>  
2  
3 using namespace std;  
4  
5 int main()  
6 {  
7     // Qui si scrivono le istruzioni del programma  
8     cout << "Ciao, mondo!" << endl;
```

```
9  
10     return 0;  
11 }
```

Se il codice sorgente è salvato nel file `hello.cpp`, lo si compila e lo si esegue con i comandi

```
g++ -Wall -std=c++17 -o hello hello.cpp  
./hello
```

e l'output è

Ciao, mondo!

Negli esempi seguenti scriviamo solo le istruzioni che contano: a meno che non contengano un programma completo, i frammenti di codice vanno inseriti fra le parentesi graffe di `main`.

Ogni istruzione di un codice sorgente C++ deve terminare con il simbolo riservato “;”.

6.2 Variabili e tipi di dato

Il codice sorgente opera su **variabili**, e ogni variabile ha un **tipo di dato**.

- Le informazioni sono classificate in diversi **tipi di dato**. Le quattro famiglie fondamentali di tipi di dato sono:
 1. Il tipo di dato “**int**” rappresenta un numero intero, cioè un numero scritto senza la virgola decimale. Comprende i numeri interi positivi e negativi, e lo zero.
 2. Il tipo di dato “**bool**” rappresenta un valore booleano: `true` o `false`.
 3. Il tipo di dato “**float**” rappresenta un numero in virgola mobile, che comprende i numeri scritti con la virgola decimale. Il tipo di dato “**double**” rappresenta un numero in virgola mobile a precisione doppia, che offre maggiore accuratezza e un intervallo più ampio di `float`.
 4. Il tipo di dato “**char**” rappresenta un singolo carattere, come una lettera, una cifra o un simbolo, racchiuso fra apici singoli (ad esempio `'A'`, `'7'`, `'!'`).
- Una sequenza di caratteri racchiusa fra virgolette doppie (ad esempio `"Ada"`) è memorizzata in una variabile di tipo “**string**”, fornito dalla libreria standard `<string>`.
- Una variabile funziona come un contenitore che memorizza un valore di un tipo di dato specifico per un uso successivo. Ogni variabile ha un nome univoco e contiene un solo valore in ogni istante.
- Alle variabili si assegnano valori con l'operatore di assegnamento “`=`”. Quando si assegna un nuovo valore, il valore precedente viene sovrascritto.

Le variabili devono essere dichiarate prima dell'uso, specificandone il tipo di dato.

Esempio 6.3: dichiarazioni di variabili e assegnamenti di valori

```

1 int age = 25; // Variabile intera
2 bool isRaining = false; // Variabile booleana
3 float smallNumber = 3.14; // Variabile in virgola mobile
4 double largeNumber = 123456.789; // Variabile in virgola mobile a doppia precisione
5 char character = 'a'; // Variabile carattere
6 string name = "Ada"; // Variabile stringa
7
8 age = 26; // Il valore precedente 25 viene sovrascritto
9 cout << age << " " << isRaining << " " << smallNumber << " "
10    << largeNumber << " " << character << " " << name << endl;

```

Output:

26 0 3.14 123457 a Ada

Il valore booleano `false` viene visualizzato come 0, e il numero 123456.789 viene visualizzato come 123457: per impostazione predefinita, `cout` visualizza al più sei cifre significative (il valore memorizzato nella variabile non cambia, e il numero di cifre visualizzate può essere scelto, come vediamo ora).

6.3 Flusso di ingresso e uscita

Per visualizzare sullo schermo i valori delle variabili, un programma usa la libreria del **flusso di ingresso e uscita**.

- L'istruzione di inclusione `"#include <iostream>"` permette di usare in C++ la libreria del flusso di ingresso e uscita.

Esempio 6.4: inclusione

```

1 #include <iostream>

```

- I valori delle variabili possono essere visualizzati come testo in uscita sullo schermo usando il flusso di uscita `"std::cout"`. L'operatore di inserimento `"<<"` serve a inviare i dati al flusso di uscita.
- In una sola istruzione `cout` si possono visualizzare più valori concatenandoli con l'operatore `"<<"`. Il testo va racchiuso fra virgolette doppie `" "`, mentre i valori delle variabili si scrivono direttamente.
- Il manipolatore `"std::endl"` si usa con `std::cout` per andare a capo sullo schermo e svuotare il buffer di uscita.

Esempio 6.5: stampe sullo schermo

```

1 int n = 7;
2 std::cout << "Il valore della variabile è " << n << std::endl;

```

Output:

Il valore della variabile è 7

- L'istruzione “`using namespace std;`” semplifica l'uso delle funzioni della libreria standard, eliminando la necessità di anteporre `std::`. D'ora in poi la usiamo sempre, e scriviamo `cout` e `endl`.
- L'istruzione di inclusione “`#include <iomanip>`” fornisce i manipolatori “`fixed`” e “`setprecision(k)`”. Quando sono scritti insieme in un'istruzione `cout`, visualizzano i numeri in virgola mobile che seguono con esattamente k cifre dopo la virgola decimale.

6.4 Operatori

Gli operatori sono simboli che eseguono operazioni su variabili e valori. In base alla loro funzione si classificano in cinque categorie principali: operatori aritmetici, di assegnamento, di incremento e decremento, relazionali e logici.

6.4.1 Operatori aritmetici

Operatori aritmetici, usati per eseguire le operazioni matematiche di base:

- `+` (addizione) $\rightsquigarrow$ somma due valori.
- `-` (sottrazione) $\rightsquigarrow$ sottrae un valore da un altro.
- `*` (moltiplicazione) $\rightsquigarrow$ moltiplica due valori.
- `/` (divisione) $\rightsquigarrow$ divide un valore per un altro.
- `%` (modulo) $\rightsquigarrow$ restituisce il resto di una divisione.

Esempio 6.6: operatori aritmetici

```
1 int a = 10, b = 3;
2
3 cout << "Addizione: " << (a + b) << endl;
4 cout << "Sottrazione: " << (a - b) << endl;
5 cout << "Moltiplicazione: " << (a * b) << endl;
6 cout << "Divisione: " << (a / b) << endl; // Divisione intera
7 cout << "Resto: " << (a % b) << endl; // Resto della divisione
```

Output:

Addizione: 13
 Sottrazione: 7
 Moltiplicazione: 30
 Divisione: 3
 Resto: 1

- L'operatore di divisione `/` esegue la divisione intera quando è usato con numeri interi: il risultato di `10 / 3` è 3. Per ottenere un risultato in virgola mobile, almeno un operando deve essere di tipo `float` o `double`. Ad esempio, `10.0 / 3` restituisce 3.33333 invece di 3.
- Un valore di tipo `int` viene convertito in un valore di tipo `double` scrivendo “(double)” davanti ad esso (**conversione di tipo**). Ad esempio, se `a` e `b` sono variabili di tipo `int`, l'espressione `(double)a / b` esegue la divisione in virgola mobile.

Esempio 6.7: divisione intera e divisione in virgola mobile

```

1 int a = 10, b = 3;
2
3 cout << a / b << endl;           // Divisione intera: 3
4 cout << 10.0 / 3 << endl;        // Divisione in virgola mobile
5 cout << (double)a / b << endl;    // Conversione di a in double
6 cout << fixed << setprecision(2) << (double)a / b << endl; // Due cifre decimali

```

Output (l'ultima riga richiede `#include <iomanip>`):

```

3
3.33333
3.33333
3.33

```

6.4.2 Operatori di assegnamento

Operatori di assegnamento, usati per assegnare valori alle variabili:

- `=` (assegnamento) $\rightsquigarrow$ assegna un valore a una variabile.
- `+=` (assegnamento con addizione) $\rightsquigarrow$ somma un valore a una variabile e assegna il risultato.
- `-=` (assegnamento con sottrazione) $\rightsquigarrow$ sottrae un valore da una variabile e assegna il risultato.
- `*=` (assegnamento con moltiplicazione) $\rightsquigarrow$ moltiplica una variabile per un valore e assegna il risultato.
- `/=` (assegnamento con divisione) $\rightsquigarrow$ divide una variabile per un valore e assegna il risultato.
- `%=` (assegnamento con modulo) $\rightsquigarrow$ applica l'operazione di modulo e assegna il risultato.

Esempio 6.8: operatori di assegnamento

```

1 int x = 10; // Assegnamento
2 cout << "Valore iniziale: " << x << endl;
3

```

```

4 x += 5; // Equivale a: x = x + 5
5 cout << "Dopo += 5: " << x << endl;
6
7 x -= 3; // Equivale a: x = x - 3
8 cout << "Dopo -= 3: " << x << endl;
9
10 x *= 2; // Equivale a: x = x * 2
11 cout << "Dopo *= 2: " << x << endl;
12
13 x /= 4; // Equivale a: x = x / 4
14 cout << "Dopo /= 4: " << x << endl;
15
16 x %= 3; // Equivale a: x = x % 3
17 cout << "Dopo %= 3: " << x << endl;

```

Output:

```

Valore iniziale: 10
Dopo += 5: 15
Dopo -= 3: 12
Dopo *= 2: 24
Dopo /= 4: 6
Dopo %= 3: 0

```

6.4.3 Operatori di incremento e decremento**Operatori di incremento e decremento:**

- ++ (incremento) $\rightsquigarrow$ aumenta di 1 il valore di una variabile.
- -- (decremento) $\rightsquigarrow$ diminuisce di 1 il valore di una variabile.
- L'operatore di incremento ++ si può usare in due forme:
 - pre-incremento (“++x”): incrementa il valore prima di restituirlo;
 - post-incremento (“x++”): restituisce prima il valore, poi lo incrementa.

La stessa logica vale per l'operatore di decremento (– -x e x- -).

Esempio 6.9: operatori di incremento e decremento

```

1 int x = 5;
2
3 cout << "Valore iniziale: " << x << endl; // Stampa 5
4
5 cout << "Pre-incremento (++x): " << ++x << endl; // Stampa 6
6
7 cout << "Post-incremento (x++): " << x++ << endl; // Stampa 6
8 cout << "Valore dopo il post-incremento: " << x << endl; // Stampa 7
9
10 cout << "Pre-decremento (--x): " << --x << endl; // Stampa 6
11

```

```

12 cout << "Post-decremento (x--): " << x-- << endl; // Stampa 6
13 cout << "Valore dopo il post-decremento: " << x << endl; // Stampa 5

```

Output:

```

Valore iniziale: 5
Pre-incremento (++x): 6
Post-incremento (x++): 6
Valore dopo il post-incremento: 7
Pre-decremento (--x): 6
Post-decremento (x--): 6
Valore dopo il post-decremento: 5

```

6.4.4 Operatori relazionali

Operatori relazionali, usati per confrontare valori e restituire un risultato booleano:

- `==` (uguale a) $\rightsquigarrow$ verifica se due valori sono uguali.
 - `!=` (diverso da) $\rightsquigarrow$ verifica se due valori sono diversi.
 - `<` (minore di) $\rightsquigarrow$ verifica se un valore è minore di un altro.
 - `>` (maggiore di) $\rightsquigarrow$ verifica se un valore è maggiore di un altro.
 - `<=` (minore o uguale a) $\rightsquigarrow$ verifica se un valore è minore o uguale a un altro.
 - `>=` (maggiore o uguale a) $\rightsquigarrow$ verifica se un valore è maggiore o uguale a un altro.
- Gli operatori relazionali restituiscono un risultato booleano:
 - `true` (visualizzato come 1) se la condizione è soddisfatta;
 - `false` (visualizzato come 0) se la condizione non è soddisfatta.

Questi operatori sono usati comunemente nelle **istruzioni condizionali** e nei **cicli**.

Esempio 6.10: operatori relazionali

```

1 int a = 10, b = 5;
2
3 cout << "a == b: " << (a == b) << endl; // Uguale a
4 cout << "a != b: " << (a != b) << endl; // Diverso da
5 cout << "a < b: " << (a < b) << endl;   // Minore di
6 cout << "a > b: " << (a > b) << endl;   // Maggiore di
7 cout << "a <= b: " << (a <= b) << endl; // Minore o uguale a
8 cout << "a >= b: " << (a >= b) << endl; // Maggiore o uguale a

```

Output:

```

a == b: 0
a != b: 1
a < b: 0

```

```

a > b: 1
a <= b: 0
a >= b: 1

```

6.4.5 Operatori logici

Operatori logici, usati per le operazioni logiche, spesso nelle istruzioni condizionali:

- `&&` (AND logico) $\leadsto$ restituisce **true** se entrambe le condizioni sono vere.
 - `||` (OR logico) $\leadsto$ restituisce **true** se almeno una condizione è vera.
 - `!` (NOT logico) $\leadsto$ inverte il valore booleano.
- Come gli operatori relazionali, gli operatori logici restituiscono valori booleani, **true** (1) o **false** (0), e sono usati comunemente nelle **istruzioni condizionali** e nei **cicli**.

Esempio 6.11: operatori logici

```

1 bool condition1 = true;
2 bool condition2 = false;
3
4 cout << "condition1: " << condition1 << endl;
5 cout << "condition2: " << condition2 << endl;
6
7 // AND logico: vero solo se entrambe le condizioni sono vere
8 cout << "condition1 && condition2: " << (condition1 && condition2) << endl;
9
10 // OR logico: vero se almeno una condizione è vera
11 cout << "condition1 || condition2: " << (condition1 || condition2) << endl;
12
13 // NOT logico: inverte il valore booleano
14 cout << "!condition1: " << !condition1 << endl;
15 cout << "!condition2: " << !condition2 << endl;

```

Output:

```

condition1: 1
condition2: 0
condition1 && condition2: 0
condition1 || condition2: 1
!condition1: 0
!condition2: 1

```

6.5 Istruzioni condizionali

Gli operatori relazionali e logici producono le condizioni che decidono quali istruzioni vengono eseguite.

- Le istruzioni condizionali permettono di eseguire blocchi di codice diversi in base a condizioni specifiche. L'istruzione `if` è la struttura di controllo fondamentale di questo tipo.
- L'istruzione `if` valuta una condizione scritta fra parentesi tonde `()`. Se la condizione è **vera**, viene eseguito il blocco di codice fra le parentesi graffe `{}`. Se la condizione è **falsa**, il programma salta il blocco ed esegue il blocco di codice fra le parentesi graffe `{}` della parte `else`, se presente.

Esempio 6.12: istruzione condizionale

```
1 int number = 10;
2
3 // Controlla se il numero è positivo
4 if (number > 0)
5 {
6     cout << "Il numero è positivo." << endl;
7 }
8 else
9 {
10     cout << "Il numero non è positivo." << endl;
11 }
```

Output:

Il numero è positivo.

6.5.1 Operatore ternario (operatore condizionale)

- L'operatore ternario `?:` offre un modo compatto per scrivere semplici istruzioni `if-else`.
- È l'unico operatore di `C++` che ha tre operandi.
- Sintassi: condizione ? valore_se_vera : valore_se_falsa
- Se la condizione è **vera**, l'espressione vale `valore_se_vera`; altrimenti vale `valore_se_falsa`.

Esempio 6.13: operatore ternario

```
1 int number = 10;
2
3 // Con l'istruzione if-else
4 string result1;
5 if (number > 0)
```

```

6     result1 = "positivo";
7 else
8     result1 = "non positivo";
9
10    cout << "Il numero è " << result1 << endl;
11
12    // Con l'operatore ternario (forma compatta)
13    string result2 = (number > 0) ? "positivo" : "non positivo";
14
15    cout << "Il numero è " << result2 << endl;

```

Output:

Il numero è positivo
Il numero è positivo

Esempio 6.14: calcolo del massimo fra due numeri

```

1  int a = 15, b = 20;
2
3  // Con l'istruzione if-else
4  int max1;
5  if (a > b)
6      max1 = a;
7  else
8      max1 = b;
9
10  cout << "Massimo (if-else): " << max1 << endl;
11
12  // Con l'operatore ternario (forma compatta)
13  int max2 = (a > b) ? a : b;
14
15  cout << "Massimo (ternario): " << max2 << endl;

```

Output:

Massimo (if-else): 20
Massimo (ternario): 20

Esempio 6.15: verifica se un numero è pari o dispari

```

1  int n = 7;
2
3  // Con l'operatore ternario
4  string parity = (n % 2 == 0) ? "pari" : "dispari";
5
6  cout << n << " è " << parity << endl;

```

Output:

7 è dispari

Esempio 6.16: assegnamento del valore assoluto

```

1 int x = -5;
2
3 // Con l'operatore ternario per calcolare il valore assoluto
4 int absValue = (x >= 0) ? x : -x;
5
6 cout << "Il valore assoluto di " << x << " è " << absValue << endl;

```

Output:

Il valore assoluto di -5 è 5

Esempio 6.17: operatori ternari annidati

```

1 int score = 85;
2
3 // Operatori ternari annidati per assegnare un voto
4 string grade = (score >= 90) ? "A" :
5                 (score >= 80) ? "B" :
6                 (score >= 70) ? "C" :
7                 (score >= 60) ? "D" : "F";
8
9 cout << "Punteggio: " << score << ", Voto: " << grade << endl;

```

Output:

Punteggio: 85, Voto: B

Le condizioni sono valutate dall'alto verso il basso: `score >= 90` è falsa e `score >= 80` è vera, quindi il voto è "B". Per la leggibilità, le condizioni complesse è meglio gestirle con istruzioni `if-else`.

6.6 Cicli

Le istruzioni condizionali scelgono *se* un blocco di codice viene eseguito; i cicli decidono *quante volte* viene eseguito.

- I cicli servono a eseguire ripetutamente un blocco di codice finché una condizione è soddisfatta. Sono essenziali per automatizzare i compiti ripetitivi e ridurre il codice ridondante. In C++, i cicli più usati sono il ciclo `while` e il ciclo `for`.

6.6.1 Il ciclo while

- Il ciclo **while** esegue un blocco di codice finché la condizione fra parentesi tonde () vale **vero**. Quando la condizione è falsa, il ciclo termina.

Esempio 6.18: ciclo while

```
1 int i = 1; // Inizializzazione
2
3 while (i <= 5) // Condizione
4 {
5     cout << "Numero: " << i << endl;
6     i++; // Incremento
7 }
```

Output:

```
Numero: 1
Numero: 2
Numero: 3
Numero: 4
Numero: 5
```

La condizione `i <= 5` viene valutata sei volte: cinque volte è vera, e la sesta volta (`i` uguale a 6) è falsa e il ciclo termina.

6.6.2 Il ciclo for

- Il ciclo **for** è una struttura di ciclo compatta che combina in un'unica istruzione inizializzazione, condizione e incremento (o decremento).

Esempio 6.19: ciclo for

```
1 for (int i = 1; i <= 5; i++) // Inizializzazione, condizione, incremento
2 {
3     cout << "Numero: " << i << endl;
4 }
```

L'output è lo stesso del ciclo **while** dell'Esempio 6.18.

6.6.3 Le istruzioni break e continue

- L'istruzione **break** termina immediatamente il ciclo.
- L'istruzione **continue** salta il resto dell'iterazione corrente e passa alla successiva.

Esempio 6.20: break e continue

```

1 for (int i = 1; i <= 10; i++)
2 {
3     if (i % 2 == 0)
4     {
5         continue; // Salta i numeri pari
6     }
7     if (i > 7)
8     {
9         break; // Interrompe il ciclo al primo numero dispari maggiore di 7
10    }
11    cout << i << " ";
12 }
13 cout << endl;

```

Output:

1 3 5 7

I valori pari di *i* sono saltati da **continue**; quando *i* è uguale a 9 l'istruzione **break** termina il ciclo, e il valore 10 non viene mai raggiunto.

6.7 Array

I cicli diventano davvero utili quando scorrono collezioni di valori. Una collezione di variabili dello stesso tipo di dato è memorizzata in un **array**.

6.7.1 Array di dimensione fissa

- Un array i cui valori sono noti quando si scrive il programma può essere dichiarato elencando i suoi valori fra parentesi graffe. La dimensione dell'array è il numero di valori elencati, e non può cambiare durante l'esecuzione.
- Ogni elemento di un array è identificato dal suo **indice**, che parte da zero, scritto fra parentesi quadre "`[]`": gli elementi di un array di dimensione *n* sono `b[0]`, `b[1]`, ..., `b[n-1]`.

Esempio 6.21: array di dimensione fissa

```

1 int b[] = {9, 12, 3, 16}; // Array di 4 interi
2 int n = 4;
3
4 b[2] = 5; // Assegna un nuovo valore al terzo elemento
5
6 for (int i = 0; i < n; i++)
7 {

```

```

8     cout << b[i] << " ";
9 }
10 cout << endl;

```

Output:

9 12 5 16

- Un array bidimensionale di dimensione fissa si dichiara con due dimensioni, il numero di righe e il numero di colonne, e i suoi valori si elencano riga per riga: `int c[2][3] = {{3, 1, 4}, {1, 5, 9}}`; dichiara un array con 2 righe e 3 colonne, e `c[1][2]` è l'elemento della seconda riga e terza colonna, uguale a 9.

6.7.2 Array di dimensione dinamica

- Un **array di dimensione dinamica** è una collezione di variabili dello stesso tipo di dato, memorizzate in locazioni di memoria contigue. La dimensione dell'array viene determinata durante l'esecuzione dell'algoritmo.
- Il simbolo riservato “*” si usa per dichiarare un **puntatore**, che memorizza l'indirizzo di memoria del primo elemento di un array di dimensione dinamica. L'operatore **new** si usa per allocare la memoria di un array di dimensione dinamica, e deve specificare il tipo di dato degli elementi insieme alla dimensione dell'array fra parentesi quadre “[]”.
- Ogni elemento di un array di dimensione dinamica è identificato dal suo indice, che parte da zero, scritto fra parentesi quadre “[]”. Un array di dimensione dinamica richiede la deallocazione della memoria con l'operatore **delete**[], per liberare la memoria ed evitare perdite di memoria (*memory leak*).

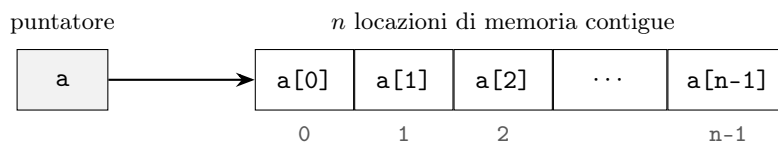


Figura 6.1: Un array di dimensione dinamica n : il puntatore a memorizza l'indirizzo del primo elemento, e gli elementi sono indicizzati da 0 a $n - 1$.

Esempio 6.22: array di dimensione dinamica

```

1  int n = 5; // Dimensione dell'array
2
3  int* a = new int[n]; // Allocazione di un array di dimensione n
4
5  for (int i = 0; i < n; i++)
6  {
7      a[i] = i * 2; // Esempio di assegnamento di valori
8  }
9
10 for (int i = 0; i < n; i++)
11 {
12     cout << a[i] << " ";

```

```

13 }
14 cout << endl;
15
16 delete[] a; // Libera la memoria allocata

```

Output:

0 2 4 6 8

- Gli array di dimensione dinamica possono essere multidimensionali. Ad esempio, il simbolo riservato “**” si usa per dichiarare un puntatore a puntatore, che memorizza l’indirizzo di memoria del primo elemento di un array bidimensionale di dimensione dinamica: un array di m puntatori, ciascuno dei quali punta a una riga di n elementi.
- L’operatore `new` si usa anche per allocare la memoria degli array bidimensionali di dimensione dinamica: prima l’array degli m puntatori alle righe, poi ogni riga. Ogni elemento di un array bidimensionale di dimensione dinamica è identificato da due indici, entrambi a partire da zero, scritti fra parentesi quadre “[] []”. La memoria viene liberata in ordine inverso: prima ogni riga, poi l’array dei puntatori alle righe.

Esempio 6.23: array bidimensionali di dimensione dinamica

```

1 int m = 3; // Numero di righe
2 int n = 4; // Numero di colonne
3
4 int** matrix = new int*[m]; // Allocazione di una matrice di dimensione m per n
5 for (int i = 0; i < m; i++)
6 {
7     matrix[i] = new int[n]; // Alloca ogni riga
8 }
9
10 for (int i = 0; i < m; i++)
11 {
12     for (int j = 0; j < n; j++)
13     {
14         matrix[i][j] = i * j; // Esempio di assegnamento di valori
15         cout << matrix[i][j] << " ";
16     }
17     cout << endl;
18 }
19
20 for (int i = 0; i < m; i++) // Libera la memoria allocata
21 {
22     delete[] matrix[i]; // Libera ogni riga
23 }
24 delete[] matrix; // Libera i puntatori alle righe

```

Output:

0 0 0 0
0 1 2 3
0 2 4 6

6.8 La funzione main e le funzioni definite dall'utente

Come nello pseudocodice, un algoritmo è scritto come una funzione che riceve l'input e restituisce l'output.

- Le funzioni permettono di scrivere codice modulare e riutilizzabile raggruppando le istruzioni in un'unica unità. Una funzione può essere chiamata più volte per eseguire lo stesso compito, riducendo la ridondanza e migliorando l'organizzazione del codice.
- Di solito le funzioni restituiscono un valore e accettano più parametri in ingresso. Quando si chiama una funzione, gli argomenti vengono passati ai parametri.
- Una funzione in C++ è costituita da:
 1. un **tipo di ritorno**: specifica il tipo del valore restituito dalla funzione (`void` se non viene restituito nulla);
 2. un **nome della funzione**: l'identificatore univoco della funzione;
 3. una **lista dei parametri** (facoltativa): specifica i valori in ingresso della funzione;
 4. un **corpo**: contiene il blocco di codice eseguito quando la funzione viene chiamata;
 5. un'istruzione di ritorno (facoltativa): serve a restituire un valore se la funzione restituisce un valore.
- La funzione `main` è il punto di ingresso di un programma C++. Viene eseguita per prima quando il programma inizia l'esecuzione. Ogni programma C++ deve contenere esattamente una funzione `main`. Il suo valore di ritorno `0` comunica al sistema operativo che il programma è terminato correttamente.
- Una funzione deve essere definita (o dichiarata) prima del punto in cui viene chiamata: per questo motivo le funzioni definite dall'utente si scrivono prima di `main`.
- Il codice sorgente C++ può anche essere compilato ed eseguito online, ad esempio al seguente indirizzo: <https://www.mycompiler.io/it/online-c++-compiler>

Esempio 6.24: funzioni

```
1 #include <iostream>
2
3 using namespace std;
4
5 // Definizione della funzione
6 int addition(int a, int b)
7 {
8     return a + b; // Restituisce la somma
9 }
10
11 // Definizione della funzione
12 int multiplication(int a, int b)
13 {
14     return a * b; // Restituisce il prodotto
15 }
16
```

```

17 int main()
18 {
19     int sum = addition(5, 10); // Chiamata della funzione
20     cout << "Somma: " << sum << endl;
21     int prod = multiplication(5, 10); // Chiamata della funzione
22     cout << "Prodotto: " << prod << endl;
23
24     return 0;
25 }

```

Output:

Somma: 15
Prodotto: 50

6.8.1 Parametri passati per valore e per riferimento

Una funzione restituisce al più un valore. Quando una funzione deve produrre più di un risultato, oppure deve modificare una variabile del chiamante, i suoi parametri vengono passati per riferimento.

- Per impostazione predefinita, un parametro è passato **per valore**: la funzione riceve una copia dell'argomento, e le modifiche fatte alla copia non influenzano la variabile del chiamante.
- Un parametro è passato **per riferimento** scrivendo il simbolo riservato "&" dopo il suo tipo di dato (ad esempio `int& x`): il parametro è un altro nome per la variabile del chiamante, e ogni modifica fatta dalla funzione è visibile al chiamante.
- Un array è passato a una funzione tramite il puntatore al suo primo elemento (ad esempio `int *a`), insieme alla sua dimensione: la funzione lavora direttamente sugli elementi dell'array del chiamante.

Esempio 6.25: parametri passati per valore e per riferimento

```

1 #include <iostream>
2
3 using namespace std;
4
5 void incrementByValue(int x)
6 {
7     x = x + 1; // Modifica solo la copia locale
8 }
9
10 void incrementByReference(int& x)
11 {
12     x = x + 1; // Modifica la variabile del chiamante
13 }
14
15 void fillWithZeros(int n, int* a)
16 {
17     for (int i = 0; i < n; i++)
18     {

```

```

19     a[i] = 0; // Modifica l'array del chiamante
20 }
21 }
22
23 int main()
24 {
25     int k = 5;
26     incrementByValue(k);
27     cout << "Dopo incrementByValue: " << k << endl;
28     incrementByReference(k);
29     cout << "Dopo incrementByReference: " << k << endl;
30
31     int b[] = {4, 7, 1};
32     fillWithZeros(3, b);
33     cout << "Array: " << b[0] << " " << b[1] << " " << b[2] << endl;
34
35     return 0;
36 }

```

Output:

```

Dopo incrementByValue: 5
Dopo incrementByReference: 6
Array: 0 0 0

```

6.9 Un algoritmo completo: il valore massimo in un array

Mettiamo ora insieme tutti i pezzi: una funzione, un ciclo, un'istruzione condizionale e un array di dimensione dinamica.

Consideriamo il problema di trovare il valore massimo in un array a di $n \geq 1$ numeri interi.

- Questo è lo pseudocodice dell'algoritmo **FindMaximum** del capitolo «Pseudocodice»: le componenti sono indicizzate da 1 a n .

Algorithm: FindMaximum

Input : Un array $a[1..n]$ di $n \geq 1$ numeri interi

Output: Il valore massimo nell'array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3   for  $i := 2$  to  $n$  do
4     if  $a[i] > max$  then
5        $max \leftarrow a[i]$ 
6     end
7   end
8   return  $max$ 
9 end

```

- Questa è l'implementazione in C++: le componenti sono indicizzate da 0 a $n - 1$, quindi l'inizializzazione usa $a[0]$ e il ciclo parte da $i = 1$. La funzione assume $n \geq 1$, come fa lo pseudocodice: per $n = 0$ l'elemento $a[0]$ non esiste.

```

1  #include <iostream>
2
3  using namespace std;
4
5  // Funzione che trova il valore massimo di un array
6  int ArrayMaxValue(int n, int *a)
7  {
8      int max = a[0]; // Inizializza con il primo elemento
9
10     for (int i = 1; i < n; i++)
11     {
12         if (a[i] > max)
13         {
14             max = a[i]; // Aggiorna il massimo
15         }
16     }
17
18     return max;
19 }
20
21 int main()
22 {
23     int n = 7; // Dimensione dell'array
24
25     int* a = new int[n]; // Allocazione di un array di dimensione n
26
27     int values[] = {9, 12, 3, 16, 19, 11, 12};
28     for (int i = 0; i < n; i++)
29     {
30         a[i] = values[i];
31     }
32
33     int max = ArrayMaxValue(n, a);
34     cout << "Valore massimo: " << max << endl;
35
36     delete[] a; // Libera la memoria allocata
37
38     return 0;
39 }

```

Output:

Valore massimo: 19

Esempio 6.26: esecuzione passo per passo

In ogni tabella, la colonna i è l'indice del ciclo C++ e la colonna $i+1$ è l'indice corrispondente dello pseudocodice. La prima riga è l'inizializzazione $\text{max} = a[0]$.

Istanza 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i (C++)	$i + 1$ (pseudocodice)	$a[i]$	$a[i] > \text{max?}$	max
—	—	—	—	9
1	2	12	Sì	12
2	3	3	No	12
3	4	16	Sì	16
4	5	19	Sì	19
5	6	11	No	19
6	7	12	No	19

Return: max = 19

Istanza 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i (C++)	$i + 1$ (pseudo-code)	$a[i]$	$a[i] > \text{max?}$	max
—	—	—	—	3
1	2	7	Sì	7
2	3	2	No	7
3	4	9	Sì	9
4	5	1	No	9

Return: max = 9

6.10 Operatori bit a bit

I numeri interi gestiti da un programma sono memorizzati in binario, come descritto nel capitolo «Rappresentazione dei numeri». C++ fornisce operatori che agiscono direttamente su queste cifre binarie.

- Gli **operatori bit a bit** eseguono operazioni sui singoli bit dei valori interi. Sono essenziali per la programmazione a basso livello, per l'ottimizzazione e per lavorare con le rappresentazioni binarie.
- I bit sono numerati da destra, a partire dalla posizione 0 (il bit meno significativo).

Operatori bit a bit in C++:

- $\&$ (AND bit a bit) $\rightsquigarrow$ restituisce 1 solo se entrambi i bit corrispondenti sono 1.
- $|$ (OR bit a bit) $\rightsquigarrow$ restituisce 1 se almeno uno dei bit corrispondenti è 1.
- $\wedge$ (XOR bit a bit) $\rightsquigarrow$ restituisce 1 se i bit corrispondenti sono diversi.
- $\sim$ (NOT bit a bit) $\rightsquigarrow$ inverte tutti i bit (0 diventa 1, 1 diventa 0).
- $\ll$ (scorrimento a sinistra) $\rightsquigarrow$ sposta i bit a sinistra di un numero specificato di posizioni.
- $\gg$ (scorrimento a destra) $\rightsquigarrow$ sposta i bit a destra di un numero specificato di posizioni.

6.10.1 AND bit a bit (&)

- L'operatore AND bit a bit confronta ogni bit di due numeri e restituisce 1 solo se entrambi i bit sono 1.
- Tavola di verità dell'AND:

Bit A	Bit B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Esempio 6.27: AND bit a bit

```

1 int a = 12; // Binario: 1100
2 int b = 10; // Binario: 1010
3 int result = a & b; // Binario: 1000 (decimale: 8)
4
5 cout << "a & b = " << result << endl; // Stampa: 8

```

Operazione bit per bit:

		Decimale
a	1 1 0 0	12
b	1 0 1 0	10
a & b	1 0 0 0	8

6.10.2 OR bit a bit (|)

- L'operatore OR bit a bit confronta ogni bit di due numeri e restituisce 1 se almeno uno dei bit è 1.
- Tavola di verità dell'OR:

Bit A	Bit B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Esempio 6.28: OR bit a bit

```

1 int a = 12; // Binario: 1100
2 int b = 10; // Binario: 1010

```

```

3 int result = a | b; // Binario: 1110 (decimale: 14)
4
5 cout << "a | b = " << result << endl; // Stampa: 14

```

Operazione bit per bit:

		Decimale
a	1 1 0 0	12
b	1 0 1 0	10
a b	1 1 1 0	14

6.10.3 XOR bit a bit (^)

- L'operatore XOR bit a bit (OR esclusivo) confronta ogni bit di due numeri e restituisce 1 se i bit sono diversi.
- Tavola di verità dello XOR:

Bit A	Bit B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Esempio 6.29: XOR bit a bit

```

1 int a = 12; // Binario: 1100
2 int b = 10; // Binario: 1010
3 int result = a ^ b; // Binario: 0110 (decimale: 6)
4
5 cout << "a ^ b = " << result << endl; // Stampa: 6

```

Operazione bit per bit:

		Decimale
a	1 1 0 0	12
b	1 0 1 0	10
a ^ b	0 1 1 0	6

6.10.4 NOT bit a bit (~)

- L'operatore NOT bit a bit inverte tutti i bit di un numero (0 diventa 1, 1 diventa 0).
- Questo è un operatore unario (opera su un solo operando).
- Il risultato dipende dal numero di bit usati per rappresentare l'intero. Nell'esempio usiamo il tipo di dato `unsigned char`, che memorizza i numeri naturali da 0 a 255 con 8 bit; la conversione (`int`) visualizza il risultato come numero invece che come carattere.

Esempio 6.30: NOT bit a bit

```
1 unsigned char a = 12; // Binario (8 bit): 00001100
2 unsigned char result = ~a; // Binario: 11110011 (decimale: 243)
3
4 cout << "~a = " << (int)result << endl; // Stampa: 243
```

Operazione bit per bit (usando la rappresentazione a 8 bit):

										Decimale
a	0	0	0	0	1	1	0	0		12
~a	1	1	1	1	0	0	1	1		243

Infatti $243 = 255 - 12$: invertire tutti gli 8 bit di un numero x dà $255 - x$.

6.10.5 Scorrimento a sinistra (<<)

- L'operatore di scorrimento a sinistra sposta tutti i bit a sinistra di un numero specificato di posizioni.
- Vengono aggiunti zeri sul lato destro.
- Scorrere a sinistra di k posizioni equivale a moltiplicare per 2^k (purché il risultato sia rappresentabile dal tipo di dato).

Esempio 6.31: scorrimento a sinistra

```
1 int a = 5; // Binario: 0101
2 int result = a << 2; // Scorrimento a sinistra di 2 posizioni: 10100 (decimale: 20)
3
4 cout << "a << 2 = " << result << endl; // Stampa: 20
5 // Equivale a: 5 * 2^2 = 5 * 4 = 20
```

Operazione bit per bit:

		Decimale	Equivalente
a	0 0 0 1 0 1	5	5
a << 1	0 0 1 0 1 0	10	5×2^1
a << 2	0 1 0 1 0 0	20	5×2^2
a << 3	1 0 1 0 0 0	40	5×2^3

6.10.6 Scorrimento a destra (>>)

- L'operatore di scorrimento a destra sposta tutti i bit a destra di un numero specificato di posizioni; i bit che escono a destra vanno persi.
- Per i numeri interi non negativi, vengono aggiunti zeri sul lato sinistro.
- Per i numeri interi non negativi, scorrere a destra di k posizioni equivale alla divisione intera per 2^k , cioè a $\lfloor x/2^k \rfloor$.

Esempio 6.32: scorrimento a destra

```

1 int a = 20; // Binario: 10100
2 int result = a >> 2; // Scorrimento a destra di 2 posizioni: 00101 (decimale: 5)
3
4 cout << "a >> 2 = " << result << endl; // Stampa: 5
5 // Equivale a: 20 / 2^2 = 20 / 4 = 5

```

Operazione bit per bit:

		Decimale	Equivalente
a	0 1 0 1 0 0	20	20
a >> 1	0 0 1 0 1 0	10	$20/2^1$
a >> 2	0 0 0 1 0 1	5	$20/2^2$
a >> 3	0 0 0 0 1 0	2	$\lfloor 20/2^3 \rfloor$

Nell'ultima riga $20/2^3 = 2.5$, e lo scorrimento a destra dà la parte intera 2: il bit 1 in posizione 0 di $a \gg 2$ va perso.

6.10.7 Applicazioni pratiche

Gli operatori bit a bit offrono soluzioni brevi e veloci per compiti comuni. Negli esempi seguenti, una **maschera** è un numero la cui rappresentazione binaria ha il bit 1 solo nelle posizioni su cui vogliamo agire: la maschera $1 \ll k$ ha il bit 1 solo nella posizione k .

Esempio 6.33: verifica se un numero è pari o dispari

```
1 int n = 15;
2
3 if (n & 1)
4     cout << n << " è dispari" << endl;
5 else
6     cout << n << " è pari" << endl;
7
8 // Spiegazione: n & 1 controlla il bit meno significativo
9 // Se vale 1 il numero è dispari, se vale 0 è pari
```

Output:

15 è dispari

Esempio 6.34: scambio di due numeri senza una variabile temporanea

```
1 int a = 5, b = 9;
2
3 cout << "Prima dello scambio: a = " << a << ", b = " << b << endl;
4
5 a = a ^ b; // ora a contiene a XOR b
6 b = a ^ b; // b = (a XOR b) XOR b = a
7 a = a ^ b; // a = (a XOR b) XOR a = b
8
9 cout << "Dopo lo scambio: a = " << a << ", b = " << b << endl;
```

Output:

Prima dello scambio: a = 5, b = 9

Dopo lo scambio: a = 9, b = 5

Lo scambio funziona perché $x \wedge x$ vale 0 e $x \wedge 0$ vale x per ogni intero x , e lo XOR è commutativo e associativo.

Esempio 6.35: verifica se un bit specifico è impostato

```
1 int n = 12; // Binario: 1100
2 int position = 2; // Controlla il bit in posizione 2 (da destra, a partire da 0)
3
4 if (n & (1 << position))
5     cout << "Il bit " << position << " vale 1" << endl;
6 else
7     cout << "Il bit " << position << " vale 0" << endl;
8
```

```

9 // Spiegazione: (1 << position) crea una maschera in cui è acceso solo il bit in
   posizione position
10 // n & mask controlla se quel bit di n vale 1

```

Output:

Il bit 2 vale 1

Esempio 6.36: impostazione di un bit specifico a 1

```

1 int n = 12; // Binario: 1100
2 int position = 1; // Imposta il bit in posizione 1
3
4 n = n | (1 << position); // Pone il bit in posizione 1 a 1
5
6 cout << "Risultato: " << n << endl; // Stampa: 14 (binario: 1110)

```

Esempio 6.37: azzeramento di un bit specifico a 0

```

1 int n = 14; // Binario: 1110
2 int position = 1; // Azzerà il bit in posizione 1
3
4 n = n & ~(1 << position); // Pone il bit in posizione 1 a 0
5
6 cout << "Risultato: " << n << endl; // Stampa: 12 (binario: 1100)

```

La maschera `~(1 << position)` ha il bit 0 nella posizione data e il bit 1 in tutte le altre posizioni.

Esempio 6.38: inversione di un bit specifico

```

1 int n = 12; // Binario: 1100
2 int position = 0; // Inverte il bit in posizione 0
3
4 n = n ^ (1 << position); // Inverte il bit in posizione 0
5
6 cout << "Risultato: " << n << endl; // Stampa: 13 (binario: 1101)

```

Punti chiave del capitolo

- Un programma C++ viene compilato (ad esempio con `g++`) e la sua esecuzione parte dalla funzione `main`; ogni istruzione termina con “;”.
- Ogni variabile è dichiarata con il suo tipo di dato prima dell'uso; la divisione fra due valori `int` è la divisione intera.

- Le istruzioni condizionali (`if`, `else`, `?:`) scelgono quali istruzioni vengono eseguite; i cicli (`while`, `for`) le ripetono.
- Gli array di `C++` sono indicizzati da 0 a $n - 1$, mentre gli array dello pseudocodice sono indicizzati da 1 a n ; un array di dimensione dinamica si alloca con `new` e si libera con `delete[]`.
- Una funzione riceve i suoi parametri per valore (una copia) o per riferimento (`&`); un array è passato tramite il puntatore al suo primo elemento, insieme alla sua dimensione.
- Gli operatori bit a bit agiscono sulla rappresentazione binaria dei numeri interi: `x << k` moltiplica x per 2^k e, per $x \geq 0$, `x >> k` calcola $\lfloor x/2^k \rfloor$.

Introduzione a Python

Introduzione

Nel capitolo «Introduzione a C++» abbiamo tradotto lo pseudocodice dei nostri algoritmi in un linguaggio di programmazione compilato. In questo capitolo introduciamo un secondo linguaggio di programmazione, **Python**, che è interpretato, ha una sintassi più leggera ed è molto diffuso sia nella didattica sia nel calcolo scientifico.

Il capitolo segue lo stesso percorso del capitolo «Introduzione a C++», in modo che i due linguaggi possano essere confrontati sezione per sezione:

- le caratteristiche principali del linguaggio e la struttura di un programma;
- variabili, tipi di dato, ingresso e uscita;
- operatori, istruzioni condizionali e cicli;
- le liste, che svolgono il ruolo degli array di C++;
- le funzioni, i loro parametri e i loro valori di ritorno;
- un algoritmo completo, la ricerca del valore massimo in un array, scritto in pseudocodice e in Python;
- gli operatori bit a bit e una tabella riassuntiva che confronta Python con C++.

Ogni frammento di codice del capitolo è stato eseguito con l'interprete `python3` (Python 3.14): gli output riportati negli esempi sono quelli stampati dai programmi.

Nello pseudocodice le componenti di un array $a[1..n]$ sono indicizzate da 1 a n ; in Python, come in C++, gli elementi di una lista di dimensione n sono indicizzati da 0 a $n - 1$. La componente $a[i]$ dello pseudocodice corrisponde all'elemento $a[i-1]$ del codice Python.

7.1 Caratteristiche principali del linguaggio di programmazione

Le caratteristiche principali di Python sono le seguenti.

- Python è un linguaggio **interpretato**: i programmi sono eseguiti istruzione per istruzione dall'**interprete Python**, senza una fase di compilazione separata. Questo rende lo sviluppo e il debug rapidi e interattivi.
- Python usa la **tipizzazione dinamica**: le variabili non devono essere dichiarate con un tipo di dato. Il tipo di dato è determinato automaticamente, durante l'esecuzione, dal valore assegnato.
- Python supporta le **funzioni**, che permettono un approccio strutturato e modulare alla programmazione. Le funzioni consentono di riutilizzare il codice e di organizzarlo meglio.
- **L'indentazione è obbligatoria** in Python: la struttura a blocchi del codice (cicli, istruzioni condizionali, funzioni) è definita dal livello di indentazione, non dalle parentesi graffe. L'indentazione standard è di 4 spazi.
- In Python, un commento su una sola riga inizia con il simbolo riservato "#". Non esiste un simbolo dedicato ai commenti su più righe: un testo racchiuso fra tripli apici `"""` (oppure `'''`) è una stringa che il programma non usa, e viene comunemente usato come commento su più righe. La stessa stringa, scritta come prima riga di una funzione, è la sua **stringa di documentazione** (docstring).

Esempio 7.1: commenti in Python

Questo è un esempio di commento su una sola riga:

```
1 # Questa è una riga di commento
```

Questo è un esempio di commento su più righe:

```
1 """
2 Questo è un
3 commento su più righe
4 """
```

7.1.1 La struttura di un programma

Un programma **Python** è un file di testo (con estensione `.py`) che l'interprete esegue dalla prima riga all'ultima. A differenza di **C++**, non è richiesta una funzione `main`: le istruzioni scritte al livello più esterno del file sono eseguite in ordine.

Esempio 7.2: lo scheletro di un programma

```
1 # Qui si scrivono le istruzioni del programma
2 print("Ciao, mondo!")
```

Se il codice sorgente è salvato nel file `hello.py`, lo si esegue con il comando

```
python3 hello.py
```

e l'output è

```
Ciao, mondo!
```

Negli esempi seguenti, a meno che non contengano un programma completo, i frammenti di codice sono scritti al livello più esterno di un file ed eseguiti in questo modo.

- Un **blocco** di istruzioni (il corpo di un'istruzione condizionale, di un ciclo o di una funzione) è introdotto dal simbolo ":" alla fine della riga precedente, e tutte le sue istruzioni sono indentate dello stesso numero di spazi. Il blocco termina alla prima istruzione indentata di meno.

Esempio 7.3: blocchi definiti dall'indentazione

```
1 x = 7
2 if x > 0:
3     print("positivo")           # Dentro il blocco dell'if
4     print("ancora dentro")     # Dentro il blocco dell'if
5 print("fuori dal blocco")     # Eseguita in ogni caso
```

Output:

```
positivo
ancora dentro
fuori dal blocco
```

In **C++** lo stesso blocco sarebbe racchiuso fra parentesi graffe `{}`; in **Python** un'indentazione sbagliata cambia il significato del programma o produce un `IndentationError`.

In **Python** le istruzioni non terminano con ";", e i blocchi sono delimitati dall'indentazione (4 spazi) invece che dalle parentesi graffe.

7.2 Variabili e tipi di dato

Il codice sorgente opera su **variabili**, e ogni valore ha un **tipo di dato**.

- Le informazioni sono classificate in diversi **tipi di dato**. I quattro tipi di dato fondamentali sono:
 - Il tipo di dato “**int**” rappresenta un numero intero, cioè un numero scritto senza la virgola decimale. Comprende i numeri interi positivi e negativi, e lo zero. A differenza di C++, gli interi di Python non hanno un valore massimo: possono avere tutte le cifre che servono.
 - Il tipo di dato “**bool**” rappresenta un valore booleano: **True** o **False** (si noti l’iniziale maiuscola).
 - Il tipo di dato “**float**” rappresenta un numero in virgola mobile, che comprende i numeri scritti con la virgola decimale. Un **float** di Python ha la stessa precisione (doppia) di un **double** di C++.
 - Il tipo di dato “**str**” rappresenta una stringa di caratteri, racchiusa fra apici singoli o doppi (ad esempio ‘**ciao**’, “**mondo**”). Non esiste un tipo di dato separato per un singolo carattere: un carattere è una stringa di lunghezza 1.
- Una variabile funziona come un contenitore che memorizza un valore. Ogni variabile ha un nome univoco.
- Alle variabili si assegnano valori con l’operatore di assegnamento “**=**”. Quando si assegna un nuovo valore, il valore precedente viene sovrascritto.
- Non serve alcuna dichiarazione** prima dell’uso: una variabile viene creata nel momento in cui le si assegna un valore. La funzione **type()** restituisce il tipo di dato di un valore.

Le variabili sono create tramite assegnamento; non c’è una fase di dichiarazione separata, e il tipo di dato non è scritto nel codice.

Esempio 7.4: assegnamenti di variabili

```

1 age = 25                                # Variabile intera
2 is_raining = False                      # Variabile booleana
3 small_number = 3.14                     # Variabile in virgola mobile
4 large_number = 123456.789                # Variabile in virgola mobile
5 character = 'a'                          # Variabile stringa (un solo carattere)
6 name = "Alice"                          # Variabile stringa
7
8 age = 26 # Il valore precedente 25 viene sovrascritto
9 print(age, is_raining, small_number, large_number, character, name)
10 print(type(age), type(is_raining), type(small_number), type(name))

```

Output:

```
26 False 3.14 123456.789 a Alice
```

```
<class 'int'> <class 'bool'> <class 'float'> <class 'str'>
```

Si confronti con l'Esempio 7.6 più avanti e con l'esempio corrispondente del capitolo «Introduzione a C++»: il valore booleano è visualizzato come **False**, e il numero 123456.789 è visualizzato con tutte le sue cifre.

Esempio 7.5: tipizzazione dinamica e interi grandi

```
1 x = 5
2 print(x, type(x))
3 x = "cinque" # Lo stesso nome ora si riferisce a una stringa
4 print(x, type(x))
5
6 print(2 ** 100) # Un intero con 31 cifre
```

Output:

```
5 <class 'int'>
cinque <class 'str'>
1267650600228229401496703205376
```

L'operatore ****** è l'elevamento a potenza (Sezione 7.4.1). In C++ il valore 2^{100} non entra in una variabile di tipo **int**.

7.3 Ingresso e uscita

- La funzione **print()** visualizza un output sullo schermo. Più valori possono essere stampati in una sola chiamata separandoli con virgole: sono visualizzati separati da uno spazio, e la riga termina con un a capo (lo stesso effetto di **endl** in C++).
- I parametri opzionali **sep** e **end** cambiano il separatore fra i valori e il testo scritto alla fine: **print(x, end=" ")** scrive uno spazio invece dell'a capo, così la **print** successiva continua sulla stessa riga.
- Le stringhe e i valori delle variabili possono essere combinati usando le **f-string** (stringhe formattate), precedute dalla lettera **f**. Dentro le f-string i nomi delle variabili (e, più in generale, le espressioni) si scrivono fra parentesi graffe **{}**.
- Dentro le parentesi graffe di una f-string, il formato **:.kf** visualizza un numero in virgola mobile con esattamente **k** cifre dopo la virgola decimale (lo stesso effetto di **fixed** e **setprecision(k)** in C++).

Esempio 7.6: stampe sullo schermo

```
1 n = 42
2 print("Il valore della variabile è", n)
3 print(f"Il valore della variabile è {n}") # Versione con f-string
4 print(f"n = {n}, doppio = {n * 2}")     # Espressione dentro {}
```

```

5
6 print("a", "b", "c", sep="-")    # Separatore "-" invece dello spazio
7 print("Niente a capo", end=" ") # Uno spazio invece dell'a capo
8 print("dopo end")
9 print(f"{10 / 3:.2f}")          # Due cifre dopo la virgola decimale

```

Output:

```

Il valore della variabile è 42
Il valore della variabile è 42
n = 42, doppio = 84
a-b-c
Niente a capo dopo end
3.33

```

- La funzione `input()` visualizza un messaggio, attende che l'utente scriva una riga di testo e la restituisce. Restituisce sempre una `str`; per ottenere un numero, il risultato deve essere convertito con `int()` o `float()`.

Esempio 7.7: ingresso da tastiera

```

1 name = input("Inserisci il tuo nome: ")    # Legge una stringa
2 n = int(input("Inserisci un intero: "))     # Legge e converte in int
3 x = float(input("Inserisci un numero reale: ")) # Legge e converte in float
4 print(f"Ciao {name}, n = {n}, x = {x}")

```

Esempio di esecuzione (l'utente scrive Ada, 7 e 2.5):

```

Inserisci il tuo nome: Ada
Inserisci un intero: 7
Inserisci un numero reale: 2.5
Ciao Ada, n = 7, x = 2.5

```

7.4 Operatori

Gli operatori sono simboli che eseguono operazioni su variabili e valori. Come in C++, sono classificati, secondo la loro funzione, in operatori aritmetici, di assegnamento, relazionali e logici; Python non ha operatori di incremento e decremento.

7.4.1 Operatori aritmetici

Operatori aritmetici, usati per eseguire le operazioni matematiche di base:

- `+` (Addizione) $\rightsquigarrow$ Somma due valori.
- `-` (Sottrazione) $\rightsquigarrow$ Sottrae un valore da un altro.
- `*` (Moltiplicazione) $\rightsquigarrow$ Moltiplica due valori.
- `/` (Divisione) $\rightsquigarrow$ Divide un valore per un altro (restituisce sempre un `float`).
- `//` (Divisione intera) $\rightsquigarrow$ Restituisce il quoziente intero (divisione con arrotondamento per difetto).
- `%` (Resto) $\rightsquigarrow$ Restituisce il resto di una divisione.
- `**` (Elevamento a potenza) $\rightsquigarrow$ Eleva un valore a una potenza.

Esempio 7.8: operatori aritmetici

```

1 a = 10
2 b = 3
3
4 print("Addizione:", a + b)
5 print("Sottrazione:", a - b)
6 print("Moltiplicazione:", a * b)
7 print("Divisione:", a / b)           # Sempre un float
8 print("Divisione intera:", a // b)
9 print("Resto:", a % b)
10 print("Elevamento a potenza:", a ** b)

```

Output:

```

Addizione: 13
Sottrazione: 7
Moltiplicazione: 30
Divisione: 3.3333333333333335
Divisione intera: 3
Resto: 1
Elevamento a potenza: 1000

```

L'ultima cifra 5 della divisione non è un errore del programma: $10/3$ ha infinite cifre decimali, e viene visualizzato il `float` più vicino.

- L'operatore di divisione `/` restituisce sempre un `float` in Python 3, anche quando divide due interi. Per ottenere un risultato intero si usa l'operatore di divisione intera `//`. È la convenzione opposta rispetto a C++, dove $10 / 3$ è la divisione intera.
- L'operatore `//` calcola $\lfloor a/b \rfloor$, arrotondando per difetto. Per operandi non negativi coincide con la divisione intera di C++; per un quoziente negativo non coincide, perché C++ tronca verso lo zero: in Python $-7 // 2$ vale -4 e $-7 \% 2$ vale 1 , mentre in C++ $-7 / 2$ vale -3 e

$-7 \% 2$ vale -1 . In entrambi i linguaggi vale l'identità $a = b \cdot q + r$ fra quoziente q e resto r .

- Un valore è convertito in un altro tipo di dato (**conversione di tipo**) con le funzioni `int()`, `float()` e `str()`: `int(3.9)` vale 3 (la parte decimale è eliminata), `float(7)` vale 7.0, `int("12")` è l'intero 12.

Esempio 7.9: divisione intera e in virgola mobile

```

1 a = 10
2 b = 3
3
4 print(a / b)           # Divisione in virgola mobile
5 print(a // b)          # Divisione intera
6 print(f"{a / b:.2f}")  # Due cifre decimali
7 print(-7 // 2, -7 % 2) # Divisione per difetto con quoziente negativo
8 print(int(-7 / 2))     # Troncamento verso zero, come in C++
9 print(int(3.9), float(7), int("12") + 1)
```

Output:

```

3.3333333333333335
3
3.33
-4 1
-3
3 7.0 13
```

7.4.2 Operatori di assegnamento

Operatori di assegnamento, usati per assegnare valori alle variabili:

- `=` (Assegnamento) $\rightsquigarrow$ Assegna un valore a una variabile.
- `+=` (Assegnamento con addizione) $\rightsquigarrow$ Somma un valore e assegna il risultato.
- `-=` (Assegnamento con sottrazione) $\rightsquigarrow$ Sottrae un valore e assegna il risultato.
- `*=` (Assegnamento con moltiplicazione) $\rightsquigarrow$ Moltiplica e assegna il risultato.
- `/=` (Assegnamento con divisione) $\rightsquigarrow$ Divide e assegna il risultato (un `float`).
- `//=` (Assegnamento con divisione intera) $\rightsquigarrow$ Divide per difetto e assegna il risultato.
- `%=` (Assegnamento con resto) $\rightsquigarrow$ Calcola il resto e assegna il risultato.
- `**=` (Assegnamento con elevamento a potenza) $\rightsquigarrow$ Eleva a una potenza e assegna il risultato.

Esempio 7.10: operatori di assegnamento

```

1 x = 10  # Assegnamento
2 print("Valore iniziale:", x)
3
4 x += 5   # Equivalente a: x = x + 5
5 print("Dopo += 5:", x)
6
7 x -= 3   # Equivalente a: x = x - 3
8 print("Dopo -= 3:", x)
9
10 x *= 2   # Equivalente a: x = x * 2
11 print("Dopo *= 2:", x)
12
13 x //= 4   # Equivalente a: x = x // 4
14 print("Dopo //= 4:", x)
15
16 x %= 4    # Equivalente a: x = x % 4
17 print("Dopo %= 4:", x)
18
19 x **= 3   # Equivalente a: x = x ** 3
20 print("Dopo **= 3:", x)
21
22 x /= 4    # Equivalente a: x = x / 4 (il risultato è un float)
23 print("Dopo /= 4:", x)

```

Output:

```

Valore iniziale: 10
Dopo += 5: 15
Dopo -= 3: 12
Dopo *= 2: 24
Dopo //= 4: 6
Dopo %= 4: 2
Dopo **= 3: 8
Dopo /= 4: 2.0

```

7.4.3 Incremento e decremento

- Python non ha gli operatori ++ e --: una variabile si incrementa con `x += 1` e si decrementa con `x -= 1`.
- L'istruzione `x++` è un errore di sintassi. L'espressione `++x` è accettata, ma non incrementa `x`: applica due volte il più unario, quindi il suo valore è `x` stesso.

Esempio 7.11: incremento e decremento

```

1 x = 5
2 x += 1      # Incremento
3 print("Dopo x += 1:", x)
4 x -= 1      # Decremento

```

```

5 print("Dopo x -= 1:", x)
6 print("++x vale", ++x, "e x vale ancora", x) # ++x non è un incremento

```

Output:

```

Dopo x += 1: 6
Dopo x -= 1: 5
++x vale 5 e x vale ancora 5

```

7.4.4 Operatori relazionali

Operatori relazionali, usati per confrontare valori e restituire un risultato booleano:

- `==` (Uguale a) $\rightsquigarrow$ Verifica se due valori sono uguali.
 - `!=` (Diverso da) $\rightsquigarrow$ Verifica se due valori sono diversi.
 - `<` (Minore di) $\rightsquigarrow$ Verifica se un valore è minore di un altro.
 - `>` (Maggiore di) $\rightsquigarrow$ Verifica se un valore è maggiore di un altro.
 - `<=` (Minore o uguale a) $\rightsquigarrow$ Verifica se un valore è minore o uguale a un altro.
 - `>=` (Maggiore o uguale a) $\rightsquigarrow$ Verifica se un valore è maggiore o uguale a un altro.
- Gli operatori relazionali restituiscono un risultato booleano: **True** se la condizione è soddisfatta, **False** altrimenti (visualizzati come **True** e **False**, non come 1 e 0 come in C++). Questi operatori sono usati comunemente nelle **istruzioni condizionali** e nei **cicli**.
 - I confronti possono essere concatenati: `0 < x <= 10` significa `0 < x and x <= 10`.

Esempio 7.12: operatori relazionali

```

1 a = 10
2 b = 5
3
4 print("a == b:", a == b) # Uguale a
5 print("a != b:", a != b) # Diverso da
6 print("a < b:", a < b)   # Minore di
7 print("a > b:", a > b)   # Maggiore di
8 print("a <= b:", a <= b) # Minore o uguale a
9 print("a >= b:", a >= b) # Maggiore o uguale a
10 print("0 < b <= a:", 0 < b <= a) # Confronto concatenato

```

Output:

```

a == b: False
a != b: True
a < b: False
a > b: True
a <= b: False
a >= b: True

```

```
0 < b <= a: True
```

7.4.5 Operatori logici

Operatori logici, usati per le operazioni logiche, spesso nelle istruzioni condizionali:

- **and** (AND logico) $\rightsquigarrow$ Restituisce **True** se entrambe le condizioni sono **True**.
- **or** (OR logico) $\rightsquigarrow$ Restituisce **True** se almeno una condizione è **True**.
- **not** (NOT logico) $\rightsquigarrow$ Inverte il valore booleano.

- Gli operatori logici di Python sono parole, mentre in C++ sono i simboli `&&`, `||` e `!`.

Esempio 7.13: operatori logici

```
1 condition1 = True
2 condition2 = False
3
4 print("condition1:", condition1)
5 print("condition2:", condition2)
6
7 # AND logico: True solo se entrambe le condizioni sono True
8 print("condition1 and condition2:", condition1 and condition2)
9
10 # OR logico: True se almeno una condizione è True
11 print("condition1 or condition2:", condition1 or condition2)
12
13 # NOT logico: inverte il valore booleano
14 print("not condition1:", not condition1)
15 print("not condition2:", not condition2)
```

Output:

```
condition1: True
condition2: False
condition1 and condition2: False
condition1 or condition2: True
not condition1: False
not condition2: True
```

7.5 Istruzioni condizionali

Gli operatori relazionali e logici producono le condizioni che decidono quali istruzioni vengono eseguite.

- Le istruzioni condizionali permettono di eseguire blocchi di codice diversi in base a condizioni specifiche.
- L'istruzione `if` valuta una condizione. Se la condizione è `True`, il blocco indentato viene eseguito.
- La clausola opzionale `elif` (else if) verifica una condizione aggiuntiva quando le precedenti sono `False`.
- La clausola opzionale `else` viene eseguita quando tutte le condizioni precedenti sono `False`.
- **Non servono parentesi** attorno alla condizione, e **non si usano parentesi graffe**: la riga della condizione termina con `:` e il blocco è delimitato dall'indentazione.

Esempio 7.14: istruzione condizionale

```
1 number = 10
2
3 # Verifica se il numero è positivo
4 if number > 0:
5     print("Il numero è positivo.")
6 else:
7     print("Il numero non è positivo.")
```

Output:

Il numero è positivo.

Esempio 7.15: if / elif / else

```
1 score = 85
2
3 if score >= 90:
4     grade = "A"
5 elif score >= 80:
6     grade = "B"
7 elif score >= 70:
8     grade = "C"
9 elif score >= 60:
10    grade = "D"
11 else:
12    grade = "F"
13
14 print(f"Punteggio: {score}, Voto: {grade}")
```

Output:

Punteggio: 85, Voto: B

Le condizioni sono valutate dall'alto verso il basso, e viene eseguito solo il blocco della prima condizione vera: `score >= 90` è falsa e `score >= 80` è vera, quindi il voto è "B". Il risultato è lo stesso degli operatori ternari annidati del capitolo «Introduzione a C++».

7.5.1 Espressione condizionale

- L'**espressione condizionale** è la versione Python dell'operatore ternario `?:` di C++: offre un modo compatto per scrivere semplici istruzioni `if-else`.
- Sintassi: `valore_se_vero if condizione else valore_se_falso`
- Se la condizione è `True`, l'espressione vale `valore_se_vero`; altrimenti vale `valore_se_falso`.

Esempio 7.16: espressione condizionale

```

1 number = 10
2
3 # Espressione condizionale compatta
4 result = "positivo" if number > 0 else "non positivo"
5 print(f"Il numero è {result}")
6
7 # Ricerca del massimo fra due numeri
8 a, b = 15, 20 # Assegnamento multiplo: a = 15 e b = 20
9 maximum = a if a > b else b
10 print(f"Massimo: {maximum}")
11
12 # Verifica se pari o dispari
13 n = 7
14 parity = "pari" if n % 2 == 0 else "dispari"
15 print(f"{n} è {parity}")
16
17 # Valore assoluto
18 x = -5
19 abs_value = x if x >= 0 else -x
20 print(f"Il valore assoluto di {x} è {abs_value}")

```

Output:

Il numero è positivo
 Massimo: 20
 7 è dispari
 Il valore assoluto di -5 è 5

La libreria di Python fornisce anche le funzioni `max(a, b)` e `abs(x)`, che restituiscono gli stessi valori.

7.6 Cicli

Le istruzioni condizionali scelgono *se* un blocco di codice viene eseguito; i cicli decidono *quante volte* viene eseguito.

- I cicli servono a eseguire ripetutamente un blocco di codice. In Python i due costrutti di ciclo sono il ciclo `while` e il ciclo `for`.

7.6.1 Il ciclo while

- Il ciclo `while` esegue un blocco di codice finché la condizione vale `True`. Se la condizione è `False` fin dall'inizio, il blocco non viene mai eseguito.

Esempio 7.17: ciclo while

```

1 i = 1                # Inizializzazione
2
3 while i <= 5:        # Condizione
4     print(f"Numero: {i}")
5     i += 1           # Incremento (in Python non esiste l'operatore ++)
```

Output:

```

Numero: 1
Numero: 2
Numero: 3
Numero: 4
Numero: 5
```

Come in C++, la condizione `i <= 5` è valutata sei volte: cinque volte è vera, e la sesta volta (`i` uguale a 6) è falsa e il ciclo termina.

7.6.2 Il ciclo for

- Il ciclo `for` di Python scorre gli elementi di un **iterabile**, come una lista, una stringa o un intervallo di interi: a ogni iterazione la variabile del ciclo assume l'elemento successivo.
- La funzione `range(start, stop, step)` genera la sequenza di interi da `start` (incluso) a `stop` (escluso), con incremento `step`. Se è dato un solo argomento, la sequenza parte da 0; se `step` è omesso, vale 1; con un `step` negativo la sequenza decresce.
- Il ciclo dello pseudocodice “`for i := 1 to n`” corrisponde a `for i in range(1, n + 1)`, e il ciclo sugli indici di una lista di dimensione n è `for i in range(n)`, con $i = 0, 1, \dots, n-1$.

Esempio 7.18: ciclo for con range

```

1 for i in range(1, 6):    # i assume i valori 1, 2, 3, 4, 5
2     print(f"Numero: {i}")
```

L'output è lo stesso del ciclo `while` dell'Esempio 7.17. La funzione `list()` mostra gli interi generati da un `range`:

```

1 print(list(range(5)))
2 print(list(range(1, 6)))
3 print(list(range(0, 10, 3)))
4 print(list(range(5, 0, -1)))

```

Output:

```

[0, 1, 2, 3, 4]
[1, 2, 3, 4, 5]
[0, 3, 6, 9]
[5, 4, 3, 2, 1]

```

Esempio 7.19: ciclo for su una lista

```

1 a = [3, 7, 2, 9, 5]
2
3 for element in a:
4     print(element)

```

Output:

```

3
7
2
9
5

```

Il ciclo visita direttamente gli elementi, senza usare i loro indici.

7.6.3 Le istruzioni break e continue

- L'istruzione **break** termina immediatamente il ciclo.
- L'istruzione **continue** salta il resto dell'iterazione corrente e passa alla successiva.

Esempio 7.20: break e continue

```

1 for i in range(1, 11):
2     if i % 2 == 0:
3         continue # Salta i numeri pari
4     if i > 7:
5         break # Interrompe il ciclo al primo dispari maggiore di 7
6     print(i, end=" ")
7 print()

```

Output:

```

1 3 5 7

```

Come nella versione C++, i valori pari di *i* sono saltati da **continue**; quando *i* è uguale a 9 l'istruzione **break** termina il ciclo, e il valore 10 non viene mai raggiunto. L'ultima **print()** scrive solo l'a capo.

7.7 Liste

I cicli diventano davvero utili quando visitano collezioni di valori. In **Python** una collezione di valori è memorizzata in una **lista**.

- Una **lista** è una collezione ordinata e modificabile di elementi, che può contenere valori di qualunque tipo. Le liste sono la principale struttura dati sequenziale di **Python** e svolgono il ruolo degli array di **C++**.
- Le liste sono racchiuse fra parentesi quadre `[]` e gli elementi sono separati da virgole. Gli elementi si raggiungono tramite il loro **indice**, a partire da 0: gli elementi di una lista **a** di dimensione n sono `a[0]`, `a[1]`, ..., `a[n-1]`. La funzione `len(a)` restituisce la dimensione della lista, e l'indice negativo `a[-1]` indica l'ultimo elemento.
- A differenza degli array di **C++**, le liste sono **dinamiche**: possono crescere o ridursi durante l'esecuzione senza gestire manualmente la memoria. L'istruzione `a.append(x)` aggiunge l'elemento `x` in fondo alla lista.
- L'espressione `[0] * n` crea una lista di n zeri. Più in generale, una **list comprehension** `[espressione for i in range(n)]` crea la lista degli n valori dell'espressione per $i = 0, 1, \dots, n-1$; quando la variabile del ciclo non è usata, per convenzione si chiama `_`.

Le liste si creano con `[]` e si indicizzano a partire da 0. Non servono né l'allocazione né la deallocazione della memoria.

Esempio 7.21: creazione e accesso a una lista

```

1 n = 5
2
3 # Crea una lista di n zeri
4 a = [0] * n
5
6 # Assegna i valori
7 for i in range(n):
8     a[i] = i * 2
9
10 print(a)          # L'intera lista
11 print(a[0])       # Primo elemento
12 print(a[4])       # Ultimo elemento, indice n-1
13 print(len(a))     # Numero di elementi

```

Output:

```

[0, 2, 4, 6, 8]
0
8
5

```

Esempio 7.22: lista scritta per esteso, modifica e crescita

```
1 b = [9, 12, 3, 16]    # Lista di 4 interi
2
3 b[2] = 5              # Assegna un nuovo valore al terzo elemento
4 b.append(20)          # La lista cresce: ora ha 5 elementi
5 print(b, len(b))
6 print(b[-1])          # L'ultimo elemento
7
8 squares = [i * i for i in range(6)] # List comprehension
9 print(squares)
```

Output:

```
[9, 12, 5, 16, 20] 5
20
[0, 1, 4, 9, 16, 25]
```

Accedere a un indice che non esiste, per esempio `b[5]`, ferma il programma con l'errore `IndexError: list index out of range` (in C++ lo stesso accesso non è controllato).

Esempio 7.23: lista scritta per esteso e iterazione sugli indici

```
1 a = [9, 12, 3, 16, 19, 11, 12]
2 n = len(a)
3
4 for i in range(n):
5     print(f"a[{i}] = {a[i]}")
```

Output:

```
a[0] = 9
a[1] = 12
a[2] = 3
a[3] = 16
a[4] = 19
a[5] = 11
a[6] = 12
```

7.7.1 Liste bidimensionali

- I dati bidimensionali possono essere rappresentati con una **lista di liste**. Ogni riga è a sua volta una lista, e gli elementi si raggiungono con due indici `matrix[i][j]`, entrambi a partire da 0: per esempio, se `c = [[3, 1, 4], [1, 5, 9]]`, allora `c[1][2]` è l'elemento nella seconda riga e terza colonna, uguale a 9.
- Una matrice di m righe e n colonne riempita di zeri si crea con la list comprehension `[[0] * n for _ in range(m)]`, che costruisce m righe distinte.

Esempio 7.24: lista bidimensionale (matrice)

```

1 m = 3      # Numero di righe
2 n = 2      # Numero di colonne
3
4 # Crea una matrice di zeri
5 matrix = [[0] * n for _ in range(m)]
6
7 # Assegna i valori
8 for i in range(m):
9     for j in range(n):
10         matrix[i][j] = i * j
11
12 # Stampa la matrice
13 for i in range(m):
14     for j in range(n):
15         print(f"matrix[{i}][{j}] = {matrix[i][j]}")
16 print(matrix)

```

Output:

```

matrix[0][0] = 0
matrix[0][1] = 0
matrix[1][0] = 0
matrix[1][1] = 1
matrix[2][0] = 0
matrix[2][1] = 2
[[0, 0], [0, 1], [0, 2]]

```

- L'espressione più breve `[[0] * n] * m` è sbagliata: ripete m volte la *stessa* riga, quindi una modifica a una riga compare in tutte le righe.

Esempio 7.25: un errore frequente con le liste bidimensionali

```

1 wrong = [[0] * 2] * 3      # Tre riferimenti alla stessa riga
2 wrong[0][0] = 1
3 print(wrong)
4
5 right = [[0] * 2 for _ in range(3)] # Tre righe distinte
6 right[0][0] = 1

```

```
7 print(right)
```

Output:

```
[[1, 0], [1, 0], [1, 0]]
[[1, 0], [0, 0], [0, 0]]
```

7.8 Funzioni

Come nello pseudocodice, un algoritmo è scritto come una funzione che riceve l'input e restituisce l'output.

- Le funzioni permettono di scrivere codice modulare e riutilizzabile, raggruppando le istruzioni in una sola unità. Una funzione si definisce con la parola chiave **def**, seguita dal nome della funzione e dall'elenco dei parametri fra parentesi, e termina con il simbolo ":".
- Una funzione in Python è composta da:
 1. la parola chiave **def** seguita dal **nome della funzione**;
 2. un **elenco di parametri** (opzionale): specifica i valori in ingresso, senza i loro tipi di dato;
 3. un **corpo** indentato: il blocco di codice eseguito quando la funzione viene chiamata;
 4. un'istruzione **return** (opzionale): restituisce un valore a chi chiama la funzione. Una funzione senza **return** restituisce il valore speciale **None**.
- Una funzione deve essere definita prima che venga eseguita l'istruzione che la chiama: per questo motivo le funzioni sono scritte all'inizio del file.
- Le istruzioni principali di un programma Python sono tipicamente poste al livello più esterno del file, oppure dentro un blocco protetto da `if __name__ == "__main__":`, che garantisce che il codice venga eseguito solo quando il file è eseguito direttamente (e non quando è importato come modulo da un altro file). Questo blocco svolge il ruolo della funzione **main** di C++.
- Il codice sorgente Python può anche essere eseguito online, per esempio al seguente indirizzo:
<https://www.mycompiler.io/online-python-compiler>

Esempio 7.26: funzioni

```
1 # Definizione di funzione
2 def addition(a, b):
3     return a + b # Restituisce la somma
4
5 # Definizione di funzione
6 def multiplication(a, b):
7     return a * b # Restituisce il prodotto
8
```

```

9 if __name__ == "__main__":
10     total = addition(5, 10)          # Chiamata di funzione
11     print(f"Somma: {total}")
12
13     product = multiplication(5, 10) # Chiamata di funzione
14     print(f"Prodotto: {product}")

```

Output:

Somma: 15
Prodotto: 50

Esempio 7.27: funzione che opera su una lista bidimensionale

```

1 def sum_matrix(matrix, rows, columns):
2     """Restituisce la somma di tutti gli elementi di una lista 2D."""
3     total = 0
4     for i in range(rows):
5         for j in range(columns):
6             total += matrix[i][j]
7     return total
8
9 if __name__ == "__main__":
10     m, n = 3, 2
11
12     # Costruisce la matrice con una list comprehension annidata
13     matrix = [[i * j for j in range(n)] for i in range(m)]
14
15     result = sum_matrix(matrix, m, n)
16     print(f"Somma degli elementi della matrice: {result}")

```

Output:

Somma degli elementi della matrice: 3

La matrice è $\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 0 & 2 \end{bmatrix}$, come nell'Esempio 7.24, e la somma dei suoi elementi è $0 + 0 + 0 + 1 + 0 + 2 = 3$.

7.8.1 Parametri e valori di ritorno

In C++ i parametri sono passati per valore o per riferimento. In Python ogni parametro riceve lo stesso oggetto che è l'argomento di chi chiama, e l'effetto di una funzione su chi la chiama dipende da ciò che la funzione fa con esso.

- Se la funzione **assegna un nuovo valore** a un parametro (per esempio `x = x + 1`), cambia solo il nome locale: la variabile di chi chiama non è toccata. Per i numeri, il comportamento è lo stesso dei parametri passati per valore in C++.
- Se la funzione **modifica gli elementi** di una lista ricevuta come parametro (per esempio `a[i] = 0`), la modifica è visibile a chi chiama, come per gli array passati a una funzione in C++.
- Una funzione può restituire **più valori** contemporaneamente, separati da virgole: sono restituiti come una **tupla**, e chi chiama li assegna a più variabili con un assegnamento multiplo. In C++ lo stesso risultato richiede parametri passati per riferimento.

Esempio 7.28: parametri e valori di ritorno multipli

```

1 def increment(x):
2     x = x + 1          # Cambia solo il nome locale x
3
4 def fill_with_zeros(a):
5     for i in range(len(a)):
6         a[i] = 0       # Modifica la lista di chi chiama
7
8 def divide(n, m):
9     return n // m, n % m # Restituisce due valori (una tupla)
10
11 if __name__ == "__main__":
12     k = 5
13     increment(k)
14     print("Dopo increment:", k)
15
16     b = [4, 7, 1]
17     fill_with_zeros(b)
18     print("Lista:", b)
19
20     q, r = divide(17, 5)
21     print("Quoziente e resto:", q, r)
22     print("Tupla:", divide(17, 5))

```

Output:

```

Dopo increment: 5
Lista: [0, 0, 0]
Quoziente e resto: 3 2
Tupla: (3, 2)

```

7.9 Un algoritmo completo: il valore massimo in un array

Mettiamo ora insieme tutti i pezzi: una funzione, un ciclo, un'istruzione condizionale e una lista.

Consideriamo il problema di trovare il valore massimo in un array a di $n \geq 1$ numeri interi.

- Questo è lo pseudocodice dell'algoritmo `FindMaximum` del capitolo «Pseudocodice»: le componenti sono indicizzate da 1 a n .

Algorithm: FindMaximum

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The maximum value in the array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3   for  $i := 2$  to  $n$  do
4     if  $a[i] > max$  then
5        $max \leftarrow a[i]$ 
6     end
7   end
8   return  $max$ 
9 end

```

- Questa è l'implementazione in Python. Gli elementi sono indicizzati da 0 a $n-1$, quindi l'inizializzazione usa $a[0]$ e il ciclo `for i in range(1, len(a))` visita gli indici $1, \dots, n-1$. La dimensione n non è passata come parametro, perché la funzione la ottiene con `len(a)`. La variabile si chiama `max_val` e non `max`, per non nascondere la funzione predefinita `max()`. La funzione suppone $n \geq 1$, come lo pseudocodice: per una lista vuota l'accesso ad $a[0]$ ferma il programma con un `IndexError`.

```

1 def array_max_value(a):
2     """Restituisce il valore massimo della lista a (len(a) >= 1)."""
3     max_val = a[0] # Inizializza con il primo elemento
4
5     for i in range(1, len(a)):
6         if a[i] > max_val:
7             max_val = a[i] # Aggiorna il massimo
8
9     return max_val
10
11 if __name__ == "__main__":
12     a = [9, 12, 3, 16, 19, 11, 12]
13
14     max_val = array_max_value(a)
15     print(f"Valore massimo: {max_val}")

```

Output:

Valore massimo: 19

- L'istanza di prova può anche essere generata a caso con il modulo `random` della libreria di Python: `random.randint(1, 100)` restituisce un intero casuale fra 1 e 100 (entrambi inclusi). L'istruzione `random.seed(2026)` fissa il punto di partenza del generatore casuale, in modo che ogni esecuzione produca la stessa lista; senza di essa, ogni esecuzione produce una lista diversa.

Esempio 7.29: un'istanza casuale

```

1 import random
2
3 def array_max_value(a):
4     max_val = a[0]
5     for i in range(1, len(a)):
6         if a[i] > max_val:
7             max_val = a[i]
8     return max_val
9
10 if __name__ == "__main__":
11     random.seed(2026) # Stessa lista casuale a ogni esecuzione
12     n = 10             # Dimensione della lista
13     a = [random.randint(1, 100) for _ in range(n)]
14
15     print(a)
16     print(f"Valore massimo: {array_max_value(a)}")

```

Output:

[16, 41, 65, 66, 83, 14, 29, 77, 80, 72]

Valore massimo: 83

Esempio 7.30: esecuzione passo per passo

In ogni tabella, la colonna i è l'indice del ciclo Python e la colonna $i + 1$ è l'indice corrispondente dello pseudocodice. La prima riga è l'inizializzazione `max_val = a[0]`.

Istanza 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i (Python)	$i + 1$ (pseudocodice)	$a[i]$	$a[i] > \text{max_val}$?	max_val
—	—	—	—	9
1	2	12	Sì	12
2	3	3	No	12
3	4	16	Sì	16
4	5	19	Sì	19
5	6	11	No	19
6	7	12	No	19

Restituisce: `max_val = 19`

Istanza 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i (Python)	$i + 1$ (pseudocodice)	$a[i]$	$a[i] > \text{max_val}?$	max_val
–	–	–	–	3
1	2	7	Sì	7
2	3	2	No	7
3	4	9	Sì	9
4	5	1	No	9

Restituisce: $\text{max_val} = 9$

7.10 Operatori bit a bit

Gli operatori bit a bit del capitolo «Introduzione a C++» esistono in Python con gli stessi simboli e lo stesso significato: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (scorrimento a sinistra) e `>>` (scorrimento a destra). La funzione `bin()` restituisce la rappresentazione binaria di un intero, preceduta dal prefisso `0b`.

- Poiché gli interi di Python non hanno un numero fisso di bit, il NOT bit a bit non dipende da un tipo di dato: `~x` è sempre uguale a $-x - 1$. Per ottenere il risultato a 8 bit di C++, il risultato è combinato con la maschera `0xFF` (il numero 255, la cui rappresentazione binaria è 11111111).

Esempio 7.31: operatori bit a bit

```

1 a = 12 # Binario: 1100
2 b = 10 # Binario: 1010
3
4 print(a & b, a | b, a ^ b)           # AND, OR, XOR
5 print(bin(a), bin(b), bin(a & b))   # Rappresentazioni binarie
6 print(5 << 2, 20 >> 2, 20 >> 3)     # Scorrimenti
7 print(~a, ~a & 0xFF)               # NOT, e la sua versione a 8 bit
8
9 n = 15
10 print("dispari" if n & 1 else "pari") # Bit meno significativo

```

Output:

```

8 14 6
0b1100 0b1010 0b1000
20 5 2
-13 243
dispari

```

Tutti i valori coincidono con quelli degli esempi corrispondenti del capitolo «Introduzione a C++».

7.11 Python e C++ a confronto

La tabella seguente riassume le principali differenze fra i due linguaggi incontrate in questo capitolo.

	C++	Python
Esecuzione	compilato (g++), poi eseguito	interpretato (python3)
Tipi di dato	dichiarati: <code>int x = 5;</code>	non dichiarati: <code>x = 5</code>
Fine dell'istruzione	<code>;</code>	fine della riga
Blocchi	parentesi graffe <code>{}</code>	<code>:</code> e indentazione
Commenti	<code>//</code> e <code>/* */</code>	<code>#</code> (e <code>""" """</code>)
Valori booleani	<code>true</code> , <code>false</code> (stampati 1, 0)	<code>True</code> , <code>False</code>
<code>10 / 3</code>	3 (divisione intera)	3.3333333333333335
Divisione intera	<code>/</code> fra due <code>int</code>	<code>//</code>
Potenza	funzione <code>pow</code>	operatore <code>**</code>
Incremento	<code>x++</code> , <code>++x</code>	<code>x += 1</code>
Operatori logici	<code>&&</code> , <code> </code> , <code>!</code>	<code>and</code> , <code>or</code> , <code>not</code>
Operatore ternario	<code>c ? x : y</code>	<code>x if c else y</code>
Ciclo <code>for</code>	<code>for (int i = 0; i < n; i++)</code>	<code>for i in range(n):</code>
Collezioni	array, di dimensione fissa o dinamica	liste, dinamiche
Dimensione di una collezione	memorizzata in una variabile separata	<code>len(a)</code>
Più risultati	parametri per riferimento	restituzione di una tupla
Punto di ingresso	funzione <code>main</code>	livello più esterno del file

Punti chiave del capitolo

- Un programma **Python** è eseguito dall'interprete (**python3**) dalla prima riga all'ultima; le istruzioni non terminano con “;” e i blocchi sono delimitati da “:” e dall'indentazione.
- Le variabili sono create tramite assegnamento, senza dichiarare il loro tipo di dato; gli interi non hanno un valore massimo.
- L'operatore `/` dà sempre un **float**; la divisione intera è `//` e arrotonda per difetto, anche per i numeri negativi.
- Le istruzioni condizionali (**if**, **elif**, **else**, e l'espressione condizionale) scelgono quali istruzioni vengono eseguite; i cicli (**while**, e **for** su un **range** o su una lista) le ripetono.
- Le liste di **Python** sono indicizzate da 0 a $n-1$, con $n = \text{len}(a)$, mentre gli array dello pseudocodice sono indicizzati da 1 a n ; le liste crescono con **append** e non richiedono gestione della memoria.
- Una funzione modifica la lista di chi chiama quando ne cambia gli elementi, ma non un numero che riceve; più risultati sono restituiti insieme come una tupla.

Introduzione a Java

Introduzione

Nei capitoli «Introduzione a C++» e «Introduzione a Python» abbiamo tradotto lo pseudocodice dei nostri algoritmi in un linguaggio di programmazione compilato e in uno interpretato. In questo capitolo introduciamo un terzo linguaggio di programmazione, **Java**, che sta a metà strada: i suoi programmi sono compilati in un codice intermedio, che viene poi eseguito da una macchina virtuale. La sua sintassi è molto vicina a quella di C++, ma ogni programma è organizzato in **classi**.

Il capitolo segue lo stesso percorso dei due capitoli precedenti, in modo che i tre linguaggi possano essere confrontati sezione per sezione:

- le caratteristiche principali del linguaggio e la struttura di un programma;
- variabili, tipi di dato, ingresso e uscita;
- operatori, istruzioni condizionali e cicli;
- gli array, e le liste dinamiche della classe `ArrayList`;
- i metodi (il nome che **Java** dà alle funzioni), i loro parametri e i loro valori di ritorno;
- un algoritmo completo, la ricerca del valore massimo in un array, scritto in pseudocodice e in **Java**;
- gli operatori bit a bit e una tabella riassuntiva che confronta **Java** con C++ e Python.

Ogni frammento di codice del capitolo è stato compilato con il compilatore `javac` ed eseguito con la macchina virtuale `java` (`OpenJDK 25`): gli output riportati negli esempi sono quelli stampati dai programmi.

Nello pseudocodice le componenti di un array $a[1..n]$ sono indicizzate da 1 a n ; in **Java**, come in C++ e in Python, gli elementi di un array di dimensione n sono indicizzati da 0 a $n - 1$. La componente $a[i]$ dello pseudocodice corrisponde all'elemento $a[i-1]$ del codice Java.

8.1 Caratteristiche principali del linguaggio di programmazione

Le caratteristiche principali di Java sono le seguenti.

- Java è un linguaggio **compilato e interpretato**: il codice sorgente (file `.java`) viene prima compilato in **bytecode** (file `.class`) dal compilatore **Java javac**, e poi il bytecode viene eseguito dalla **Java Virtual Machine (JVM)**. Questo rende i programmi **Java indipendenti dalla piattaforma**: lo stesso bytecode funziona su qualunque macchina su cui sia installata una JVM («scrivi una volta, esegui ovunque»).
- Java è un linguaggio **fortemente e staticamente tipizzato**: ogni variabile deve essere dichiarata con il suo tipo di dato prima dell'uso, e il suo tipo di dato non può cambiare durante l'esecuzione.
- Java è **orientato agli oggetti**: ogni programma è organizzato in **classi**. Il punto di ingresso di ogni programma Java è il metodo `main` di una classe.
- Java supporta i **metodi** (il termine Java per le funzioni), che permettono un approccio modulare e riutilizzabile alla programmazione.
- Come in C++, la struttura a blocchi del codice è definita dalle **parentesi graffe {}**, e ogni istruzione deve terminare con il simbolo riservato `“;”`.
- Come in C++, un commento su una sola riga inizia con il simbolo riservato `“//”`, e i commenti su più righe sono racchiusi fra i simboli riservati `“/*”` e `“*/”`.

Esempio 8.1: commenti in Java

Questo è un esempio di commento su una sola riga:

```
1 // Questa è una riga di commento
```

Questo è un esempio di commento su più righe:

```
1 /* Questo è un  
2    commento su più righe */
```

8.1.1 La struttura di un programma

Prima di descrivere le singole istruzioni, osserviamo lo scheletro che ogni programma ha in comune. Un programma Java è costituito da una o più classi; l'esecuzione parte dal metodo `public static void main(String[] args)` della classe che viene lanciata.

Esempio 8.2: lo scheletro di un programma

```

1 public class HelloWorld {
2
3     public static void main(String[] args) {
4         // Qui si scrivono le istruzioni del programma
5         System.out.println("Ciao, mondo!");
6     }
7 }

```

Una classe dichiarata `public` deve essere salvata in un file con lo stesso nome della classe: qui `HelloWorld.java`. Il programma si compila e si esegue con i comandi

```

javac HelloWorld.java
java HelloWorld

```

Il primo comando produce il file di bytecode `HelloWorld.class`; il secondo lo esegue (si noti che l'estensione non si scrive). L'output è

```
Ciao, mondo!
```

Negli esempi seguenti scriviamo solo le istruzioni che contano: a meno che non contengano un programma completo, i frammenti di codice vanno inseriti fra le parentesi graffe di `main`.

Ogni istruzione Java appartiene a un metodo, e ogni metodo appartiene a una classe. Ogni istruzione termina con “;”.

8.2 Variabili e tipi di dato

Il codice sorgente opera su **variabili**, e ogni variabile ha un **tipo di dato**.

- I quattro tipi di dato fondamentali di Java sono:
 1. Il tipo di dato “`int`” rappresenta un intero memorizzato con 32 bit: un numero intero da $-2^{31} = -2147483648$ a $2^{31} - 1 = 2147483647$. Per interi più grandi si usa il tipo di dato `long` (64 bit).
 2. Il tipo di dato “`boolean`” rappresenta un valore booleano: `true` o `false` (minuscolo).
 3. Il tipo di dato “`double`” rappresenta un numero in virgola mobile a precisione doppia (64 bit), come in C++. Il tipo di dato `float` ha precisione singola, e i suoi valori si scrivono con il suffisso `f` (ad esempio `3.14f`).
 4. Il tipo di dato “`char`” rappresenta un singolo carattere, racchiuso fra apici singoli (ad esempio `'A'`, `'7'`).
- Il tipo di dato “`String`” (con la `S` maiuscola) rappresenta una sequenza di caratteri racchiusa fra virgolette doppie, ed è il modo standard di trattare il testo in Java.

- Una variabile funziona come un contenitore che memorizza un valore di un tipo di dato specifico per un uso successivo.
- Alle variabili si assegnano valori con l'operatore di assegnamento "=". Quando si assegna un nuovo valore, il valore precedente viene sovrascritto.
- Il compilatore controlla i tipi di dato: assegnare un valore di tipo sbagliato (per esempio la stringa "cinque" a una variabile di tipo `int`) o usare una variabile che non ha ricevuto un valore è un **errore di compilazione**, e il programma non viene prodotto.

Le variabili devono essere dichiarate con il loro tipo di dato prima dell'uso, come in C++.

Esempio 8.3: dichiarazioni e assegnamenti di variabili

```

1 int age = 25;           // Variabile intera
2 boolean isRaining = false; // Variabile booleana
3 double pi = 3.14159;    // Virgola mobile a precisione doppia
4 float small = 3.14f;    // Virgola mobile a precisione singola
5 double large = 123456.789; // Virgola mobile a precisione doppia
6 char letter = 'A';      // Variabile carattere
7 String name = "Alice";  // Variabile stringa
8
9 age = 26; // Il valore precedente 25 viene sovrascritto
10 System.out.println(age + " " + isRaining + " " + pi + " " + small);
11 System.out.println(large + " " + letter + " " + name);

```

Output:

```

26 false 3.14159 3.14
123456.789 A Alice

```

A differenza di C++, il valore booleano è visualizzato come `false` e il numero 123456.789 è visualizzato con tutte le sue cifre.

Esempio 8.4: i limiti del tipo di dato `int`

```

1 int big = Integer.MAX_VALUE; // Il più grande int
2 System.out.println(big);
3 System.out.println(big + 1); // Overflow: il risultato non entra in 32 bit
4
5 long bigger = (long) big + 1; // Calcolato con 64 bit
6 System.out.println(bigger);

```

Output:

```

2147483647
-2147483648
2147483648

```

Quando il risultato di un'operazione fra valori `int` non entra in 32 bit, «riparte» dall'altro estremo dell'intervallo, senza alcun messaggio di errore. Gli interi di Python non vanno mai in overflow.

8.3 Ingresso e uscita

- L'oggetto `System.out` (l'uscita standard, fornita dalla classe `System`) ha i metodi che visualizzano valori sullo schermo. I più comuni sono:
 - `System.out.println()`: stampa un valore seguito da un a capo;
 - `System.out.print()`: stampa un valore senza l'a capo.
- Valori e stringhe sono **concatenati** con l'operatore `+` dentro un'istruzione di stampa: quando uno dei due operandi di `+` è una `String`, l'altro viene convertito in testo. L'espressione è valutata da sinistra a destra, quindi un'operazione aritmetica dentro una concatenazione deve essere racchiusa fra parentesi.
- Il metodo `System.out.printf()` stampa un testo formattato: nella prima stringa, `%d` è sostituito da un intero, `%s` da una stringa, `%.kf` da un numero in virgola mobile con esattamente k cifre dopo la virgola decimale, e `%n` da un a capo.

Esempio 8.5: stampe sullo schermo

```

1 int n = 42;
2 System.out.println("Il valore della variabile è " + n);
3 System.out.println("n = " + n + ", doppio = " + (n * 2));
4
5 System.out.println("n + 1 = " + n + 1);    // Concatenazione: "42" poi "1"
6 System.out.println("n + 1 = " + (n + 1));  // Addizione, poi concatenazione
7
8 System.out.print("Niente a capo ");
9 System.out.println("dopo print");
10 System.out.printf("%d diviso %d è %.2f%n", 10, 3, 10.0 / 3);

```

Output:

```

Il valore della variabile è 42
n = 42, doppio = 84
n + 1 = 421
n + 1 = 43
Niente a capo dopo print
10 diviso 3 è 3.33

```

- Il separatore decimale stampato da `printf` dipende dalle impostazioni della lingua del calcolatore: su un calcolatore impostato in italiano l'ultima riga è `10 diviso 3 è 3,33`, con la virgola. Il metodo `println` usa sempre il punto.
- Per leggere l'ingresso da tastiera si usa la classe `Scanner` del pacchetto `java.util`. Va importata con l'istruzione `import java.util.Scanner;`, scritta all'inizio del file, prima della classe. I metodi `nextLine()`, `nextInt()` e `nextDouble()` leggono una riga di testo, un intero e un numero in virgola mobile.

Esempio 8.6: ingresso da tastiera con Scanner

```

1 import java.util.Scanner;
2
3 public class InputExample {
4
5     public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
7
8         System.out.print("Inserisci il tuo nome: ");
9         String name = sc.nextLine();    // Legge una String
10
11        System.out.print("Inserisci un intero: ");
12        int n = sc.nextInt();           // Legge un int
13
14        System.out.print("Inserisci un numero reale: ");
15        double x = sc.nextDouble();     // Legge un double
16
17        System.out.println("Ciao " + name + ", n = " + n + ", x = " + x);
18        sc.close();
19    }
20 }

```

Esempio di esecuzione (l'utente scrive Ada, 7 e 2.5):

```

Inserisci il tuo nome: Ada
Inserisci un intero: 7
Inserisci un numero reale: 2.5
Ciao Ada, n = 7, x = 2.5

```

Come `printf`, il metodo `nextDouble()` segue le impostazioni della lingua: su un calcolatore impostato in italiano il numero reale va scritto come 2,5.

8.4 Operatori

Gli operatori sono simboli che eseguono operazioni su variabili e valori. Gli operatori di Java sono gli stessi di C++: operatori aritmetici, di assegnamento, di incremento e decremento, relazionali e logici.

8.4.1 Operatori aritmetici

Operatori aritmetici, usati per eseguire le operazioni matematiche di base:

- `+` (Addizione) $\rightsquigarrow$ Somma due valori.
- `-` (Sottrazione) $\rightsquigarrow$ Sottrae un valore da un altro.
- `*` (Moltiplicazione) $\rightsquigarrow$ Moltiplica due valori.
- `/` (Divisione) $\rightsquigarrow$ Divide un valore per un altro. Divisione intera quando entrambi gli operandi sono interi.
- `%` (Resto) $\rightsquigarrow$ Restituisce il resto di una divisione.

Esempio 8.7: operatori aritmetici

```

1 int a = 10, b = 3;
2
3 System.out.println("Addizione:      " + (a + b));
4 System.out.println("Sottrazione:    " + (a - b));
5 System.out.println("Moltiplicazione: " + (a * b));
6 System.out.println("Divisione:      " + (a / b)); // Divisione intera
7 System.out.println("Resto:          " + (a % b)); // Resto della divisione
8 System.out.println("Divisione reale: " + ((double) a / b));

```

Output:

```

Addizione:      13
Sottrazione:    7
Moltiplicazione: 30
Divisione:      3
Resto:          1
Divisione reale: 3.3333333333333335

```

- Come in C++, l'operatore di divisione / esegue la **divisione intera** quando entrambi gli operandi sono interi, e il quoziente è troncato verso lo zero: $10 / 3$ vale 3 e $-7 / 2$ vale -3 (con resto $-7 \% 2$ uguale a -1). Per ottenere un risultato in virgola mobile almeno un operando deve essere di tipo `double`, per esempio con la **conversione di tipo (cast)** `(double) a / b`.
- La conversione `(int)` elimina la parte decimale di un `double`: `(int) 3.9` vale 3. Non esiste un operatore per la potenza: il metodo `Math.pow(x, y)` restituisce x^y come `double`.

Esempio 8.8: divisione intera e in virgola mobile

```

1 int a = 10, b = 3;
2
3 System.out.println(a / b);           // Divisione intera
4 System.out.println(10.0 / 3);        // Divisione in virgola mobile
5 System.out.println((double) a / b); // Conversione di a in double
6 System.out.printf("%.2f\n", (double) a / b); // Due cifre decimali
7 System.out.println(-7 / 2 + " " + -7 % 2); // Troncamento verso zero
8 System.out.println((int) 3.9 + " " + Math.pow(2, 10));

```

Output:

```

3
3.3333333333333335
3.3333333333333335
3.33
-3 -1
3 1024.0

```

8.4.2 Operatori di assegnamento

Operatori di assegnamento, usati per assegnare valori alle variabili:

- `=` (Assegnamento) $\rightsquigarrow$ Assegna un valore a una variabile.
- `+=` (Assegnamento con addizione) $\rightsquigarrow$ Somma un valore e assegna il risultato.
- `-=` (Assegnamento con sottrazione) $\rightsquigarrow$ Sottrae un valore e assegna il risultato.
- `*=` (Assegnamento con moltiplicazione) $\rightsquigarrow$ Moltiplica e assegna il risultato.
- `/=` (Assegnamento con divisione) $\rightsquigarrow$ Divide e assegna il risultato.
- `%=` (Assegnamento con resto) $\rightsquigarrow$ Calcola il resto e assegna il risultato.

Esempio 8.9: operatori di assegnamento

```
1 int x = 10; // Assegnamento
2 System.out.println("Valore iniziale: " + x);
3
4 x += 5; System.out.println("Dopo += 5: " + x); // x = x + 5
5 x -= 3; System.out.println("Dopo -= 3: " + x); // x = x - 3
6 x *= 2; System.out.println("Dopo *= 2: " + x); // x = x * 2
7 x /= 4; System.out.println("Dopo /= 4: " + x); // x = x / 4
8 x %= 4; System.out.println("Dopo %= 4: " + x); // x = x % 4
```

Output:

```
Valore iniziale: 10
Dopo += 5: 15
Dopo -= 3: 12
Dopo *= 2: 24
Dopo /= 4: 6
Dopo %= 4: 2
```

Due istruzioni possono essere scritte sulla stessa riga, perché ciascuna termina con “;”.

8.4.3 Operatori di incremento e decremento

Operatori di incremento e decremento:

- ++ (Incremento) $\rightsquigarrow$ Aumenta di 1 il valore di una variabile.
 - -- (Decremento) $\rightsquigarrow$ Diminuisce di 1 il valore di una variabile.
- Come in C++, il pre-incremento (++x) incrementa il valore prima di restituirlo, e il post-incremento (x++) restituisce prima il valore, poi lo incrementa. La stessa logica vale per l'operatore di decremento (--x e x--).

Esempio 8.10: operatori di incremento e decremento

```

1 int x = 5;
2 System.out.println("Valore iniziale: " + x);
3 System.out.println("Pre-incremento (++x): " + ++x);
4 System.out.println("Post-incremento (x++): " + x++);
5 System.out.println("Valore dopo il post-incremento: " + x);
6 System.out.println("Pre-decremento (--x): " + --x);
7 System.out.println("Post-decremento (x--): " + x--);
8 System.out.println("Valore dopo il post-decremento: " + x);

```

Output:

```

Valore iniziale: 5
Pre-incremento (++x): 6
Post-incremento (x++): 6
Valore dopo il post-incremento: 7
Pre-decremento (--x): 6
Post-decremento (x--): 6
Valore dopo il post-decremento: 5

```

8.4.4 Operatori relazionali

Operatori relazionali, usati per confrontare valori e restituire un risultato boolean:

- == (Uguale a) $\rightsquigarrow$ Verifica se due valori sono uguali.
 - != (Diverso da) $\rightsquigarrow$ Verifica se due valori sono diversi.
 - < (Minore di) $\rightsquigarrow$ Verifica se un valore è minore di un altro.
 - > (Maggiore di) $\rightsquigarrow$ Verifica se un valore è maggiore di un altro.
 - <= (Minore o uguale a) $\rightsquigarrow$ Verifica se un valore è minore o uguale a un altro.
 - >= (Maggiore o uguale a) $\rightsquigarrow$ Verifica se un valore è maggiore o uguale a un altro.
- Gli operatori relazionali restituiscono un valore di tipo **boolean**, visualizzato come **true** o **false** (non come 1 e 0 come in C++). Questi operatori sono usati comunemente nelle **istruzioni condizionali** e nei **cicli**.

- L'operatore `==` confronta i valori dei tipi di dato primitivi (`int`, `double`, `char`, `boolean`); due valori `String` si confrontano con il metodo `equals`, per esempio `name.equals("Alice")`.

Esempio 8.11: operatori relazionali

```

1 int a = 10, b = 5;
2
3 System.out.println("a == b: " + (a == b)); // Uguale a
4 System.out.println("a != b: " + (a != b)); // Diverso da
5 System.out.println("a < b: " + (a < b));    // Minore di
6 System.out.println("a > b: " + (a > b));    // Maggiore di
7 System.out.println("a <= b: " + (a <= b)); // Minore o uguale a
8 System.out.println("a >= b: " + (a >= b)); // Maggiore o uguale a

```

Output:

```

a == b: false
a != b: true
a < b: false
a > b: true
a <= b: false
a >= b: true

```

8.4.5 Operatori logici

Operatori logici, usati per le operazioni logiche, spesso nelle istruzioni condizionali:

- `&&` (AND logico) $\rightsquigarrow$ Restituisce `true` se entrambe le condizioni sono `true`.
- `||` (OR logico) $\rightsquigarrow$ Restituisce `true` se almeno una condizione è `true`.
- `!` (NOT logico) $\rightsquigarrow$ Inverte il valore booleano.

Esempio 8.12: operatori logici

```

1 boolean c1 = true, c2 = false;
2
3 System.out.println("c1: " + c1);
4 System.out.println("c2: " + c2);
5 System.out.println("c1 && c2: " + (c1 && c2)); // true solo se entrambe sono true
6 System.out.println("c1 || c2: " + (c1 || c2)); // true se almeno una è true
7 System.out.println("!c1: " + (!c1));          // Inverte il valore
8 System.out.println("!c2: " + (!c2));

```

Output:

```

c1: true
c2: false
c1 && c2: false
c1 || c2: true
!c1: false

```

!c2: true

8.5 Istruzioni condizionali

Gli operatori relazionali e logici producono le condizioni che decidono quali istruzioni vengono eseguite.

- L'istruzione `if` valuta una condizione racchiusa fra parentesi `()`. Se la condizione è `true`, viene eseguito il blocco racchiuso fra parentesi graffe `{}`. La clausola opzionale `else if` verifica condizioni aggiuntive, e la clausola opzionale `else` tratta tutti i casi rimanenti.
- A differenza di `C++`, la condizione deve essere di tipo `boolean`: un intero non è accettato come condizione, quindi `if (n % 2)` è un errore di compilazione e il test va scritto `if (n % 2 != 0)`.

Esempio 8.13: istruzione condizionale

```
1 int number = 10;
2
3 // Verifica se il numero è positivo
4 if (number > 0) {
5     System.out.println("Il numero è positivo.");
6 } else {
7     System.out.println("Il numero non è positivo.");
8 }
```

Output:

Il numero è positivo.

Esempio 8.14: `if` / `else if` / `else`

```
1 int score = 85;
2 String grade;
3
4 if (score >= 90) { grade = "A"; }
5 else if (score >= 80) { grade = "B"; }
6 else if (score >= 70) { grade = "C"; }
7 else if (score >= 60) { grade = "D"; }
8 else { grade = "F"; }
9
10 System.out.println("Punteggio: " + score + ", Voto: " + grade);
```

Output:

Punteggio: 85, Voto: B

Le condizioni sono valutate dall'alto verso il basso, e viene eseguito solo il blocco della prima condizione vera. La variabile `grade` è dichiarata senza un valore: il compilatore la accetta perché ogni ramo le assegna un valore prima che venga stampata.

8.5.1 Operatore ternario (operatore condizionale)

- L'operatore ternario `?:` è lo stesso di C++. Sintassi: `condizione ? valore_se_vero : valore_se_falso`.

Esempio 8.15: operatore ternario

```
1 int number = 10;
2
3 // condizione ? valore_se_vero : valore_se_falso
4 String result = (number > 0) ? "positivo" : "non positivo";
5 System.out.println("Il numero è " + result);
6
7 // Ricerca del massimo fra due numeri
8 int a = 15, b = 20;
9 int maximum = (a > b) ? a : b;
10 System.out.println("Massimo: " + maximum);
11
12 // Verifica se pari o dispari
13 int n = 7;
14 String parity = (n % 2 == 0) ? "pari" : "dispari";
15 System.out.println(n + " è " + parity);
16
17 // Valore assoluto
18 int x = -5;
19 int absValue = (x >= 0) ? x : -x;
20 System.out.println("Il valore assoluto di " + x + " è " + absValue);
```

Output:

```
Il numero è positivo
Massimo: 20
7 è dispari
Il valore assoluto di -5 è 5
```

La classe `Math` fornisce anche i metodi `Math.max(a, b)` e `Math.abs(x)`, che restituiscono gli stessi valori.

8.6 Cicli

Le istruzioni condizionali scelgono *se* un blocco di codice viene eseguito; i cicli decidono *quante volte* viene eseguito.

- I cicli servono a eseguire ripetutamente un blocco di codice finché una condizione è soddisfatta. Come in C++, i cicli più usati di Java sono il ciclo `while` e il ciclo `for`.

8.6.1 Il ciclo `while`

- Il ciclo `while` esegue un blocco di codice finché la condizione racchiusa fra parentesi `()` vale `true`. Quando la condizione è `false`, il ciclo termina.

Esempio 8.16: ciclo `while`

```
1 int i = 1; // Inizializzazione
2
3 while (i <= 5) { // Condizione
4     System.out.println("Numero: " + i);
5     i++; // Incremento
6 }
```

Output:

```
Numero: 1
Numero: 2
Numero: 3
Numero: 4
Numero: 5
```

8.6.2 Il ciclo `for`

- Il ciclo `for` combina inizializzazione, condizione e incremento (o decremento) in una sola istruzione, esattamente come in C++.

Esempio 8.17: ciclo `for`

```
1 // Inizializzazione; Condizione; Incremento
2 for (int i = 1; i <= 5; i++) {
3     System.out.println("Numero: " + i);
4 }
```

L'output è lo stesso del ciclo `while` dell'Esempio 8.16.

8.6.3 Le istruzioni break e continue

- L'istruzione **break** termina immediatamente il ciclo.
- L'istruzione **continue** salta il resto dell'iterazione corrente e passa alla successiva.

Esempio 8.18: break e continue

```

1 for (int i = 1; i <= 10; i++) {
2     if (i % 2 == 0) {
3         continue; // Salta i numeri pari
4     }
5     if (i > 7) {
6         break; // Interrompe il ciclo al primo dispari maggiore di 7
7     }
8     System.out.print(i + " ");
9 }
10 System.out.println();

```

Output:

1 3 5 7

L'output è lo stesso delle versioni C++ e Python: il valore 9 fa scattare il **break**.

8.7 Array

I cicli diventano davvero utili quando visitano collezioni di valori. Una collezione di variabili dello stesso tipo di dato è memorizzata in un **array**.

- Un **array** in Java è una collezione ordinata di elementi dello stesso tipo di dato, con una dimensione fissata al momento della creazione: la dimensione non può cambiare in seguito.
- Il tipo di dato di un array di interi si scrive `int[]`. Un array si alloca con l'operatore **new**, per esempio `new int[n]`, dove la dimensione `n` può essere decisa durante l'esecuzione, come per gli array di dimensione dinamica di C++. Gli elementi di un nuovo array sono inizializzati a 0 (a `false` per `boolean`).
- Gli elementi si raggiungono tramite il loro **indice** fra parentesi quadre `[]`, a partire da 0: gli elementi di un array di dimensione `n` sono `a[0]`, `a[1]`, ..., `a[n-1]`. Ogni array conosce la propria dimensione: la lunghezza di un array `a` è `a.length`.
- A differenza di C++, la memoria di un array non deve essere liberata con `delete[]`: viene rilasciata automaticamente dal **garbage collector** della JVM quando l'array non è più usato.

Gli array si allocano con **new** prima dell'uso (oppure si creano con un elenco di valori fra parentesi graffe). Gli indici partono da 0, la dimensione è fissa ed è data da `a.length`.

Esempio 8.19: creazione e accesso a un array

```

1  int n = 5;
2
3  int[] a = new int[n]; // Allocazione di un array di n interi
4
5  for (int i = 0; i < n; i++) {
6      a[i] = i * 2; // Esempio di assegnamento di valori
7  }
8
9  System.out.println("Lunghezza: " + a.length);
10 System.out.println("a[0] = " + a[0]); // Primo elemento
11 System.out.println("a[4] = " + a[4]); // Ultimo elemento, indice n-1

```

Output:

```

Lunghezza: 5
a[0] = 0
a[4] = 8

```

Accedere a un indice che non esiste, per esempio `a[5]`, ferma il programma con l'errore `ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5` (in C++ lo stesso accesso non è controllato).

- Un array i cui valori sono noti quando si scrive il programma può essere creato elencando i suoi valori fra parentesi graffe, come in C++.
- Il **ciclo for esteso** `for (int x : a)` visita direttamente gli elementi dell'array `a`, senza usare i loro indici, come il ciclo `for` su una lista di Python.
- Il metodo `Arrays.toString(a)`, della classe `Arrays` del pacchetto `java.util`, restituisce gli elementi di un array come un testo fra parentesi quadre.

Esempio 8.20: array scritto per esteso e iterazione

```

1  import java.util.Arrays;
2
3  int[] a = {9, 12, 3, 16, 19, 11, 12};
4  int n = a.length;
5
6  for (int i = 0; i < n; i++) {      // Ciclo sugli indici
7      System.out.println("a[" + i + "] = " + a[i]);
8  }
9
10 for (int x : a) {                  // Ciclo sugli elementi
11     System.out.print(x + " ");
12 }
13 System.out.println();
14
15 System.out.println(Arrays.toString(a));

```

Output:

```

a[0] = 9

```

```

a[1] = 12
a[2] = 3
a[3] = 16
a[4] = 19
a[5] = 11
a[6] = 12
9 12 3 16 19 11 12
[9, 12, 3, 16, 19, 11, 12]

```

L'istruzione `import` si scrive all'inizio del file, prima della classe.

8.7.1 Array bidimensionali

- Gli array bidimensionali (matrici) si dichiarano con due coppie di parentesi quadre `[] []`, e si allocano con `new int[m][n]` (m righe e n colonne) in una sola istruzione. L'elemento nella riga i e colonna j si raggiunge come `matrix[i][j]`, con entrambi gli indici a partire da 0; `matrix.length` è il numero di righe e `matrix[i].length` è il numero di colonne della riga i .
- Un array bidimensionale può anche essere creato elencando i suoi valori riga per riga: `int[][] c = {{3, 1, 4}, {1, 5, 9}};` crea un array con 2 righe e 3 colonne, e `c[1][2]` è l'elemento nella seconda riga e terza colonna, uguale a 9.

Esempio 8.21: array bidimensionale (matrice)

```

1 int m = 3; // Numero di righe
2 int n = 2; // Numero di colonne
3
4 int[][] matrix = new int[m][n]; // Allocazione di una matrice m x n
5
6 for (int i = 0; i < m; i++) {
7     for (int j = 0; j < n; j++) {
8         matrix[i][j] = i * j; // Esempio di assegnamento di valori
9     }
10 }
11
12 for (int i = 0; i < m; i++) {
13     for (int j = 0; j < n; j++) {
14         System.out.println("matrix[" + i + "][" + j + "] = " + matrix[i][j]);
15     }
16 }

```

Output:

```

matrix[0][0] = 0
matrix[0][1] = 0
matrix[1][0] = 0
matrix[1][1] = 1
matrix[2][0] = 0
matrix[2][1] = 2

```

8.7.2 Liste di dimensione variabile: ArrayList

- Quando il numero di elementi non è noto in anticipo, la classe `ArrayList` del pacchetto `java.util` fornisce una lista che cresce durante l'esecuzione, come le liste di Python. Il tipo di dato di una lista di interi è `ArrayList<Integer>`: gli elementi di una `ArrayList` sono oggetti, e `Integer` è la classe che corrisponde a `int`.
- Il metodo `add(x)` aggiunge l'elemento `x` in fondo alla lista, `get(i)` restituisce l'elemento di indice `i`, `set(i, x)` lo sostituisce, e `size()` restituisce il numero di elementi.

Esempio 8.22: una lista di dimensione variabile

```

1 import java.util.ArrayList;
2
3 ArrayList<Integer> list = new ArrayList<>(); // Lista vuota
4
5 for (int i = 1; i <= 4; i++) {
6     list.add(i * i); // La lista cresce
7 }
8 list.set(0, 100);    // Sostituisce l'elemento di indice 0
9
10 System.out.println(list + " ha " + list.size() + " elementi");
11 System.out.println("list.get(2) = " + list.get(2));

```

Output:

```

[100, 4, 9, 16] ha 4 elementi
list.get(2) = 9

```

8.8 Metodi (funzioni)

Come nello pseudocodice, un algoritmo è scritto come una funzione che riceve l'input e restituisce l'output. In Java le funzioni si chiamano **metodi** e devono sempre essere definite dentro una **classe**.

- Un metodo in Java è composto da:
 1. un **modificatore di accesso** opzionale (ad esempio `public`, `private`) e la parola chiave `static` quando il metodo appartiene alla classe invece che a un oggetto della classe (tutti i metodi di questo capitolo sono `static`);
 2. un **tipo di ritorno**: specifica il tipo del valore restituito (`void` se non viene restituito nulla);
 3. un **nome del metodo**: l'identificatore univoco del metodo;
 4. un **elenco di parametri** (opzionale): specifica i valori in ingresso, ciascuno con il suo tipo di dato;
 5. un **corpo**: il blocco di codice eseguito quando il metodo viene chiamato;
 6. un'istruzione `return`, obbligatoria se il tipo di ritorno non è `void`.

- Il **punto di ingresso** di ogni programma Java è il metodo `public static void main(String[] args)`, che svolge il ruolo della funzione `main` di C++. Il parametro `args` contiene le parole scritte dopo il nome della classe nel comando `java`.
- A differenza di C++, un metodo può essere chiamato prima del punto in cui è definito nella classe: l'ordine dei metodi dentro una classe non conta.
- Il codice sorgente Java può anche essere compilato ed eseguito online, per esempio al seguente indirizzo:
<https://www.mycompiler.io/online-java-compiler>

Esempio 8.23: metodi

```
1 public class Main {  
2  
3     // Definizione di metodo  
4     static int addition(int a, int b) {  
5         return a + b; // Restituisce la somma  
6     }  
7  
8     // Definizione di metodo  
9     static int multiplication(int a, int b) {  
10        return a * b; // Restituisce il prodotto  
11    }  
12  
13    public static void main(String[] args) {  
14        int sum = addition(5, 10); // Chiamata di metodo  
15        System.out.println("Somma: " + sum);  
16  
17        int prod = multiplication(5, 10); // Chiamata di metodo  
18        System.out.println("Prodotto: " + prod);  
19    }  
20 }
```

Output:

Somma: 15
Prodotto: 50

Esempio 8.24: metodo che opera su un array bidimensionale

```
1 public class Main {  
2  
3     static int sumMatrix(int[][] matrix, int rows, int cols) {  
4         int sum = 0;  
5         for (int i = 0; i < rows; i++) {  
6             for (int j = 0; j < cols; j++) {  
7                 sum += matrix[i][j];  
8             }  
9         }  
10        return sum;  
11    }  
12 }
```

```

12
13     public static void main(String[] args) {
14         int m = 3, n = 2;
15         int[] [] matrix = new int[m][n];
16
17         for (int i = 0; i < m; i++)
18             for (int j = 0; j < n; j++)
19                 matrix[i][j] = i * j;
20
21         int total = sumMatrix(matrix, m, n);
22         System.out.println("Somma degli elementi della matrice: " + total);
23     }
24 }

```

Output:

Somma degli elementi della matrice: 3

Il corpo di un ciclo `for` fatto di una sola istruzione può essere scritto senza parentesi graffe, come in C++.

8.8.1 Parametri e valori di ritorno

- In Java ogni parametro è passato **per valore**: il metodo riceve una copia dell'argomento. Per i tipi di dato primitivi (`int`, `double`, `boolean`, `char`) le modifiche fatte alla copia non influiscono sulla variabile di chi chiama; non esiste il passaggio per riferimento con il simbolo `&` di C++.
- Una variabile di tipo `int[]` non contiene gli elementi, ma un **riferimento** all'array (simile al puntatore di C++). Il metodo riceve una copia del riferimento, che punta allo stesso array: le modifiche fatte agli *elementi* dell'array sono visibili a chi chiama.
- Un metodo restituisce al più un valore. Quando un metodo deve restituire più risultati, può restituirli in un array, per esempio `new int[]{q, r}`.

Esempio 8.25: parametri e risultati multipli

```

1 public class Main {
2
3     static void increment(int x) {
4         x = x + 1; // Modifica solo la copia locale
5     }
6
7     static void fillWithZeros(int[] a) {
8         for (int i = 0; i < a.length; i++) {
9             a[i] = 0; // Modifica l'array di chi chiama
10        }
11    }
12
13    static int[] divide(int n, int m) {
14        return new int[]{n / m, n % m}; // Due risultati in un array
15    }
16 }

```

```

17     public static void main(String[] args) {
18         int k = 5;
19         increment(k);
20         System.out.println("Dopo increment: " + k);
21
22         int[] b = {4, 7, 1};
23         fillWithZeros(b);
24         System.out.println("Array: " + b[0] + " " + b[1] + " " + b[2]);
25
26         int[] qr = divide(17, 5);
27         System.out.println("Quoziente e resto: " + qr[0] + " " + qr[1]);
28     }
29 }

```

Output:

Dopo increment: 5
 Array: 0 0 0
 Quoziente e resto: 3 2

8.9 Un algoritmo completo: il valore massimo in un array

Mettiamo ora insieme tutti i pezzi: un metodo, un ciclo, un'istruzione condizionale e un array.

Consideriamo il problema di trovare il valore massimo in un array a di $n \geq 1$ numeri interi.

- Questo è lo pseudocodice dell'algoritmo FindMaximum del capitolo «Pseudocodice»: le componenti sono indicizzate da 1 a n .

Algorithm: FindMaximum

Input : An array $a[1..n]$ of $n \geq 1$ integer numbers

Output: The maximum value in the array a

```

1 function FindMaximum( $a, n$ )
2    $max := a[1]$ 
3   for  $i := 2$  to  $n$  do
4     if  $a[i] > max$  then
5        $max \leftarrow a[i]$ 
6     end
7   end
8   return  $max$ 
9 end

```

- Questa è l'implementazione in Java. Gli elementi sono indicizzati da 0 a $n - 1$, quindi l'inizializzazione usa $a[0]$ e il ciclo parte da $i = 1$. La dimensione n non è passata come parametro, perché il metodo la ottiene con `a.length`. Il metodo suppone $n \geq 1$, come lo pseudocodice: per un array vuoto l'accesso ad $a[0]$ ferma il programma con una `ArrayIndexOutOfBoundsException`.

```

1 public class Main {
2
3     // Metodo che trova il valore massimo in un array
4     static int arrayMaxValue(int[] a) {
5         int max = a[0]; // Inizializza con il primo elemento
6
7         for (int i = 1; i < a.length; i++) {
8             if (a[i] > max) {
9                 max = a[i]; // Aggiorna il massimo
10            }
11        }
12
13        return max;
14    }
15
16    public static void main(String[] args) {
17        int[] a = {9, 12, 3, 16, 19, 11, 12};
18
19        int max = arrayMaxValue(a);
20        System.out.println("Valore massimo: " + max);
21    }
22 }

```

Output:

Valore massimo: 19

- L'istanza di prova può anche essere generata a caso con la classe `Random` del pacchetto `java.util`: se `rand` è un oggetto di questa classe, `rand.nextInt(100)` restituisce un intero casuale fra 0 e 99, quindi `rand.nextInt(100) + 1` è fra 1 e 100. L'oggetto creato con `new Random(2026)` parte dal valore fisso 2026 (il **seme**), in modo che ogni esecuzione produca lo stesso array; l'oggetto creato con `new Random()` produce un array diverso a ogni esecuzione.

Esempio 8.26: un'istanza casuale

```

1 import java.util.Arrays;
2 import java.util.Random;
3
4 public class Main {
5
6     static int arrayMaxValue(int[] a) {
7         int max = a[0];
8         for (int i = 1; i < a.length; i++) {
9             if (a[i] > max) {
10                 max = a[i];
11            }
12        }
13        return max;
14    }
15
16    public static void main(String[] args) {
17        int n = 10; // Dimensione dell'array

```

```

18     int[] a = new int[n];
19     Random rand = new Random(2026); // Stesso array casuale a ogni esecuzione
20     for (int i = 0; i < n; i++) {
21         a[i] = rand.nextInt(100) + 1; // Intero casuale fra 1 e 100
22     }
23
24     System.out.println(Arrays.toString(a));
25     System.out.println("Valore massimo: " + arrayMaxValue(a));
26 }
27 }

```

Output:

[100, 51, 98, 31, 50, 39, 74, 37, 35, 79]
 Valore massimo: 100

Esempio 8.27: esecuzione passo per passo

In ogni tabella, la colonna i è l'indice del ciclo Java e la colonna $i+1$ è l'indice corrispondente dello pseudocodice. La prima riga è l'inizializzazione $\text{max} = a[0]$.

Istanza 1: $a = [9, 12, 3, 16, 19, 11, 12]$, $n = 7$

i (Java)	$i + 1$ (pseudocodice)	$a[i]$	$a[i] > \text{max}?$	max
—	—	—	—	9
1	2	12	Sì	12
2	3	3	No	12
3	4	16	Sì	16
4	5	19	Sì	19
5	6	11	No	19
6	7	12	No	19

Restituisce: $\text{max} = 19$

Istanza 2: $a = [3, 7, 2, 9, 1]$, $n = 5$

i (Java)	$i + 1$ (pseudocodice)	$a[i]$	$a[i] > \text{max}?$	max
—	—	—	—	3
1	2	7	Sì	7
2	3	2	No	7
3	4	9	Sì	9
4	5	1	No	9

Restituisce: $\text{max} = 9$

8.10 Operatori bit a bit

Gli operatori bit a bit del capitolo «Introduzione a C++» esistono in **Java** con gli stessi simboli e lo stesso significato: `&` (AND), `|` (OR), `^` (XOR), `~` (NOT), `<<` (scorrimento a sinistra) e `>>` (scorrimento a destra). Il metodo `Integer.toBinaryString(x)` restituisce la rappresentazione binaria di un intero.

- Un valore di tipo `int` ha 32 bit, quindi `~12` inverte 32 bit e dà `-13`; il risultato a 8 bit di C++ si ottiene con la maschera `0xFF`, come in Python.
- Java ha un terzo operatore di scorrimento, `>>>` (scorrimento a destra senza segno), che aggiunge sempre zeri a sinistra, anche per i numeri negativi; per i numeri non negativi coincide con `>>`.
- Poiché una condizione deve essere di tipo `boolean`, il test del bit meno significativo si scrive `(n & 1) == 1`, e non `(n & 1)` come in C++.

Esempio 8.28: operatori bit a bit

```

1  int a = 12; // Binario: 1100
2  int b = 10; // Binario: 1010
3
4  System.out.println((a & b) + " " + (a | b) + " " + (a ^ b)); // AND, OR, XOR
5  System.out.println(Integer.toBinaryString(a & b));           // 1000
6  System.out.println((5 << 2) + " " + (20 >> 2) + " " + (20 >> 3)); // Scorrimenti
7  System.out.println(~a + " " + (~a & 0xFF)); // NOT, e la sua versione a 8 bit
8  System.out.println((-8 >> 1) + " " + (-8 >>> 28)); // Scorrimento con e senza segno
9
10 int n = 15;
11 System.out.println((n & 1) == 1 ? "dispari" : "pari"); // Bit meno significativo

```

Output:

```

8 14 6
1000
20 5 2
-13 243
-4 15
dispari

```

Il numero `-8` è memorizzato come `11...11000` (32 bit): `-8 >> 1` mantiene il segno e dà `-4`, mentre `-8 >>> 28` mantiene solo i 4 bit più a sinistra `1111`, cioè `15`.

8.11 Java, C++ e Python a confronto

La tabella seguente riassume le principali differenze fra i tre linguaggi di questa parte del corso.

	C++	Java	Python
Esecuzione	compilato (g++)	compilato in bytecode (javac), eseguito dalla JVM (java)	interpretato (python3)
Tipi di dato	dichiarati	dichiarati	non dichiarati
Blocchi	{ }	{ }	: e indentazione
Fine dell'istruzione	;	;	fine della riga
Valori booleani	true, false (stampati 1, 0)	true, false	True, False
10 / 3	3	3	3.3333333333333335
-7 / 2 (intero)	-3	-3	-7 // 2 vale -4
Intero più grande	$2^{31} - 1$ per int	$2^{31} - 1$ per int	nessun limite
Operatori logici	&&, , !	&&, , !	and, or, not
Collezioni	array, liberati con delete[]	array (a.length) e ArrayList, liberati automaticamente	liste (len(a)), liberate automaticamente
Indice fuori intervallo	non controllato	eccezione	eccezione
Più risultati	parametri per riferimento	restituzione di un array	restituzione di una tupla
Punto di ingresso	funzione main	metodo main di una classe	livello più esterno del file

Punti chiave del capitolo

- Un programma Java è costituito da classi; è compilato in bytecode con `javac` ed eseguito dalla JVM con `java`, a partire dal metodo `public static void main(String[] args)`.
- Ogni variabile è dichiarata con il suo tipo di dato, che è controllato dal compilatore; il tipo di dato `int` ha 32 bit, e la divisione fra due interi è la divisione intera.
- Le istruzioni condizionali (`if`, `else if`, `else`, `?:`) e i cicli (`while`, `for`, e il `for` esteso) hanno la stessa sintassi di C++, ma una condizione deve essere di tipo `boolean`.
- Gli array di Java si allocano con `new`, conoscono la loro dimensione `a.length`, sono indicizzati da 0 a $n - 1$ e sono liberati automaticamente; la classe `ArrayList` fornisce liste di dimensione variabile.
- I parametri sono passati per valore; un metodo che riceve un array può modificarne gli elementi, e più risultati sono restituiti insieme in un array.