Strutture di controllo, tipi di base, funzioni e array

Matteo Spanio

19 marzo 2024

In questa lezione si termina l'introduzione alla sintassi del C, con un focus sulle strutture di controllo, le espressioni, i tipi di base, le funzioni e agli array. Gli argomenti vengono presentati in ordine diverso rispetto al libro di testo.

Tenendo a mente l'architettura di Von Neumann per i calcolatori, si può riassumere che i computer hanno bisogno di 2 principali elementi: **dati** e **istruzioni** per manipolare i dati. I linguaggi di programmazione forniscono in maniera trasparente queste componenti attraverso l'uso di **variabili** ed **espressioni**.

Le espressioni sono un insieme di:

1. **variabili**,
2. **costanti**,
3. **operatori**.

Tipi di dato

Le varibili possono contenere diversi tipi di dati, i tipi di dati base in C sono:

- interi: int, short, long int, unsigned short/int/long int
- virgola mobile: float, double, long double
- char: caratteri

Inoltre esiste il tipo speciale `void` che indica l'assenza di informazione.

Tipi numerici

I vari tipi di interi e tipi di virgola mobile si distinguono per il numero di byte usati per la loro rappresentazione in memoria e, conseguentemente, per il range di valori che possono rappresentare.

La dimensione di un tipo di dato, espressa in byte, e il suo range di valori dipendono dal compilatore e dall'architettura del computer. Per determinare la dimensione di un tipo di dato si può utilizzare l'operatore sizeof, mentre per conoscere il range di valori utilizzabili si possono sfruttare le librerie `limits.h` (per i tipi interi) e `float.h` (per i tipi di virgola mobile).

```c
#include <limits.h>
#include <stdio.h>

int main(void)
{
    printf("size of short: %lu\n", sizeof(short));
    printf("size of int: %lu\n", sizeof(int));
    printf("size of double: %lu\n", sizeof(double));
    printf("INT_MAX = %d\n", INT_MAX);
    printf("INT_MIN = %d\n", INT_MIN);
    printf("UINT_MAX = %u\n", UINT_MAX);
    printf("LONG_MAX = %ld\n", LONG_MAX);
    printf("LONG_MIN = %ld\n", LONG_MIN);
    printf("ULONG_MAX = %lu\n", ULONG_MAX);
    printf("SHRT_MAX = %d\n", SHRT_MAX);
    printf("SHRT_MIN = %d\n", SHRT_MIN);
    printf("USHRT_MAX = %u\n", USHRT_MAX);
}
```

L'esempio precedente illustra come ottenere informazioni su dimensione e range di alcuni tipi numerici in C. È importante notare come la gestione attenta di questi aspetti possa prevenire errori di overflow e underflow.

Tipi char

Il tipo char è usato per rappresentare singoli caratteri. In C, i caratteri sono trattati come piccoli interi, consentendo di eseguire operazioni aritmetiche su di loro grazie alla codifica ASCII.

Esempio di dichiarazione:

```c
char lettera = 'A';
```

è essenziale usare gli apici singoli (' ') per denotare i valori di tipo char, distinguendoli dalle stringhe, che sono rappresentate tra virgolette doppie (" ").

```c
#include <stdio.h>

int main(void)
{
    char lettera_minuscola = 'a';
    char lettera_maiuscola = lettera_minuscola - 32;
    printf("La lettera maiuscola corrispondente è: %c\n", lettera_maiuscola);
    return 0;
}
```

Questo frammento di codice dimostra la conversione di una lettera minuscola in maiuscola sfruttando la differenza costante nei valori ASCII.

> **Tabella ASCII**
>
> Nella tabella ASCII, i valori dei caratteri alfabetici maiuscoli sono minori di quelli dei corrispondenti caratteri minuscoli, per trovare la distanza tra i due valori basta effettuare l'operazione `'A' - 'a'`. Questo valore è costante e può essere usato per convertire una lettera minuscola in maiuscola e viceversa. Per maggiori dettagli si veda la tabella ASCII alla pagina del manuale linux `man ascii`.

Conversioni tra tipi

In generale, per effettuare operazioni aritmetiche tra due numeri, questi devono essere dello stesso tipo.

In C è possibile mescolare tipi diversi nella stessa espressione, e il compilatore effettuerà automaticamente le conversioni necessarie per eseguire l'operazione richiesta.

Conversioni implicite

```c
int a = 5;
float b = 1.618;

a + b; // 6.618
```

In questo esempio, il valore intero a viene convertito implicitamente in virgola mobile prima di eseguire l'operazione di somma. Questo tipo di conversione è chiamato "*promotion*".

Se assegno un valore a una variabile di tipo diverso, il compilatore effettuerà una conversione implicita.

```c
int i = 1;
float f;
f = i; // 1.0
f = 2; // 2.0
```

È vero quindi anche il contrario:

```c
float f = 1.618;
int i;
i = f; // 3
```

In questo caso, il valore in virgola mobile viene troncato al valore intero più vicino, si noti che non viene effettuato alcun arrotondamento.

Cast espliciti

È possibile convertire esplicitamente il tipo di un' espressione con l'operazione di casting:

`(nome_tipo) espressione`

Il risultato dell'espressione viene convertito al tipo specificato.

Un caso frequente è quando si esegue una divisione tra due interi:

```c
float quotient;
int dividend, divisor;
quotient = (float) dividend / divisor;
```

> **Nota**
>
> In questo caso, `dividend` viene convertito in float prima di eseguire la divisione, ciò comporta che `divisor` venga convertito implicitamente in float, infatti, come vedremo nel paragrafo successivo, l'operatore di casting ha la precedenza su quello di divisione.

Operatori

Gli operatori in C sono fondamentali per eseguire operazioni su variabili e valori. Ogni operatore ha diverse caratteristiche che influenzano il modo in cui viene valutata un'espressione.

Tipi di Operatori

Gli operatori in C possono essere suddivisi in diverse categorie in base al tipo di operazioni che eseguono. Ogni operatore ha una priorità che determina l'ordine in cui viene valutato all'interno di un'espressione. Gli operatori con una priorità più alta vengono valutati prima degli operatori con una priorità più bassa. Nel caso di operatori della stessa priorità, l'associatività determina l'ordine di valutazione.

Ecco una tabella riassuntiva degli operatori C, ordinati da quelli con più alta priorità a quelli con minore priorità:

Categoria	Operatori	Associatività
Postfissi	`++ -- -> . () []`	Sinistra-Destra
Unari / Prefissi	`+ - ! ~ ++ -- (type) * & sizeof`	Destra-Sinistra
Moltiplicativi	`* / %`	Sinistra-Destra
Additivi	`+ -`	Sinistra-Destra
Shift	`<< >>`	Sinistra-Destra
Relazionali	`< > <= >=`	Sinistra-Destra
Uguaglianza	`== !=`	Sinistra-Destra
Bitwise AND	`&`	Sinistra-Destra
Bitwise XOR	`^`	Sinistra-Destra
Bitwise OR	`|`	Sinistra-Destra
Logico AND	`&&`	Sinistra-Destra
Logico OR	`||`	Sinistra-Destra
Condizionale	`?:`	Destra-Sinistra
Assegnamento	`= += -= *= /= %= <<= >>= &= ^= |=`	Destra-Sinistra
Virgola	`,`	Sinistra-Destra

Alcuni operatori sono comuni a molti linguaggi di programmazione (additivi, moltiplicativi, relazionali, uguaglianza, …), altri però acquisiscono un significato specifico in C, come ad esempio l'operatore di dereferenziazione `*` o l'operatore di indirizzamento `&`. Nel corso di questa lezione ne approfondiremo alcuni, mentre altri verranno trattati successivamente.

Side-Effects

Gli operatori di incremento e decremento (`++` e `--`) sono potenti ma vanno usati con cautela. Possono essere utilizzati sia come operazioni prefisse che postfix. La differenza principale tra queste due forme sta nel momento in cui viene effettuata l'operazione di incremento o decremento.

Nel caso dell'operatore di post-incremento `i++`, l'incremento avviene dopo che il valore di `i` viene utilizzato nell'espressione corrente. Nel caso dell'operatore di pre-incremento `++i`, l'incremento avviene prima che il valore di `i` venga utilizzato nell'espressione.

Questo può portare a comportamenti diversi, soprattutto quando si utilizzano questi operatori all'interno della stessa espressione o dello stesso statement. Ad esempio:

```
int i = 1;
printf("i is %d\n", i++);  // Stampa: i is 1
printf("i is %d\n", i);    // Stampa: i is 2
printf("i is %d\n", ++i);  // Stampa: i is 3
printf("i is %d\n", i);    // Stampa: i is 3
```

Nel primo `printf`, l'operatore post-incremento viene utilizzato, quindi il valore di `i` (1) viene stampato e poi incrementato a 2. Nel secondo `printf`, il valore incrementato di `i` (2) viene stampato. Nel terzo `printf`, l'operatore pre-incremento viene utilizzato, quindi `i` viene incrementato a 3 e poi stampato. Infine, nel quarto `printf`, viene stampato di nuovo il valore corrente di `i`, che è 3.

È importante prestare attenzione a questi comportamenti per evitare risultati inaspettati nel proprio codice.

> **Nota**
>
> Gli altri operatori di uso frequente che comportano un side effect sono quelli di assegnamento `=`, `+=`, `-=` e così via. Questi operatori modificano il valore della variabile a sinistra dell'operatore.

Espressioni istruzione

Tutte le espressioni possono essere anche *statement*: in una linea posso avere anche solo una operazione singola seguita da `;`

```
i = 1;       // utile
i++;         // utile
i * j + 2    // inutile, potrebbe dare warning
             // "statement with no effect"
```

Nella seconda riga dell'esempio il risultato viene scartato, ma la modifica avviene lo stesso.

Espressioni logiche

Alcuni operatori sono responsabili della valutazione di espressioni booleane, ossia espressioni che possono essere valutate come vere o false.

Operatori Booleani

Ecco gli operatori booleani disponibili in C:

- `&&` (AND logico)
- `||` (OR logico)
- `!` (NOT logico)
- `==` (uguaglianza)
- `!=` (diverso)
- `>` (maggiore)
- `<` (minore)
- `>=` (maggiore o uguale)
- `<=` (minore o uguale)

Questi operatori producono tutti 0 o 1 come risultato, ma in C qualsiasi valore diverso da 0 è considerato vero.

> **Avviso**
>
> È importante ricordarsi che gli operatori bitwise (`&`, `|`, `^`, `~`) non sono operatori booleani, ma operatori di manipolazione dei bit. Non vanno confusi con gli operatori booleani, pertanto non vanno utilizzati per valutare espressioni booleane.

AND Logico (`&&`)

L'operatore `&&` restituisce 1 se entrambe le espressioni booleane sono non-zero, altrimenti restituisce 0. È un operatore a corto circuito, quindi se il risultato è già noto dopo aver valutato la prima espressione, la seconda non viene valutata.

OR Logico (`||`)

L'operatore `||` restituisce 1 se almeno una delle due espressioni booleane è non-zero, altrimenti restituisce 0. Anche questo è un operatore a corto circuito.

NOT Logico (`!`)

L'operatore `!` inverte il valore di verità di un'espressione. Restituisce 1 se l'espressione è zero e 0 se l'espressione è non-zero.

Operatori di Confronto

Gli operatori di confronto (`==`, `!=`, `>`, `<`, `>=`, `<=`) confrontano due valori e restituiscono un valore intero in base alla relazione tra di essi.

> **Nota**
>
> Mentre il risultato di questi operatori è intuitivo su degli interi o float, bisogna notare che dalla parte sinistra e destra possono esserci anche `char` (che in C sono rappresentati come interi da un byte), che valore restituisce l'espressione `'Z' < 'a'`? E `':' < ';'`? Suggerimento: per rispondere senza scrivere codice si provi a consultare la pagina del manuale di `ascii` (`man ascii`).

Valutazione delle Espressioni Booleane

Le espressioni booleane vengono valutate in base alle regole della logica booleana. Il risultato di un'espressione booleana è un valore intero, dove 0 rappresenta "falso" e 1 rappresenta "vero". Ad esempio:

```
int i = 3, j = 2, k = 1;
int result = (i < j && j < k); // result sarà 0 perché entrambe le espressioni sono false
```

In questo caso, l'espressione `i < j && j < k` sarà valutata come `0 && 0`, il che restituirà 0 perché entrambe le espressioni sono false.

Il tipo bool

Come già detto, il tipo di ritorno delle espressioni logiche in C è un intero, il tipo booleano non esiste nativamente in C. Quindi, per rappresentare valori booleani, si utilizzano valori interi, dove:

- 0 → falso
- 1 → vero (e tutti gli interi diversi da 0)

> **Nota**
>
> Da C99 in poi, `bool` è un tipo definito in `stdbool.h` che può assumere solo i valori `true` e `false`.
>
> ```
> #include <stdbool.h>
> bool b = true;
> ```
>
> Si tratta di un tipo di dato intero, ma è considerato un tipo booleano. Ciò è possibile grazie all'uso di macro definite in `stdbool.h`:
>
> ```
> // nel file stdbool.h
> #define true 1
> #define false 0
> #define bool _Bool // _Bool è un tipo di dato intero introdotto in C99
> ```

Control flow

Essendo C un linguaggio strutturato, possiede un insieme di istruzioni di controllo del flusso di esecuzione del programma. Queste istruzioni permettono di eseguire un blocco di codice solo se una condizione è vera, di eseguire un blocco di codice ripetutamente, di eseguire un blocco di codice solo se una condizione è falsa, ecc. Le principali istruzioni di controllo del flusso sono:

- `if-else`
- `switch`
- `for`
- `while`
- `do-while`
- `break`
- `continue`

`if-else`

L'istruzione `if-else` permette di eseguire un blocco di codice solo se una condizione è vera. La sintassi è la seguente:

```
if (espressione) {
    // statements
} else {
    // statements
}
```

Se la condizione è vera, viene eseguito il blocco di codice immediatamente successivo all'istruzione `if`. Altrimenti, viene eseguito il blocco di codice immediatamente successivo all'istruzione `else`.

```c
#include <stdio.h>

int main() {
    int x = 10;

    if (x > 5) {
        printf("x è maggiore di 5\n");
    } else {
        printf("x è minore o uguale a 5\n");
    }

    return 0;
}
```

Il codice stampa:

```
x è maggiore di 5
```

`if-else` annidati

È possibile annidare più istruzioni `if-else` all'interno di un blocco di codice.

```c
#include <stdio.h>

int main() {
    int x = 10;

    if (x > 5) {
        if (x > 7) {
            printf("x è maggiore di 7\n");
        } else {
            printf("x è minore o uguale a 7\n");
        }
    } else {
        printf("x è minore o uguale a 5\n");
    }

    return 0;
}
```

Il codice stampa:

```
x è maggiore di 7
```

Oppure si può usare l'istruzione `else if` per evitare l'annidamento.

```c
#include <stdio.h>

int main(void)
{
    int x = 10;
    if (x > 7)
    {
        printf("x è maggiore di 7\n");
    }
    else if (x > 5)
    {
        printf("x è maggiore di 5\n");
```

```
    }
    else
    {
        printf("x è minore o uguale a 5\n");
    }
}
```

Questa versione stampa lo stesso risultato della precedente, in questo caso però, il secondo `if` è allineato con il primo `else`, rendendo il codice più leggibile.

> **Attenzione**
>
> Un errore comune è confondere l'operatore di assegnamento `=` con l'operatore di confronto `==`. Ad esempio, l'espressione `x = 10` assegna il valore `10` alla variabile `x`, mentre l'espressione `x == 10` confronta il valore della variabile `x` con `10`. Se si scrive `if (x = 10)`, l'espressione è sempre vera, perché assegna `10` alla variabile `x` e restituisce `10`, che è considerato `true`. Per confrontare il valore della variabile `x` con `10`, si deve invece scrivere `if (x == 10)`.

If ternario

L'if ternario è una forma compatta dell'istruzione `if-else`. La sintassi è la seguente:

```
espressione ? valore_se_vera : valore_se_falsa
```

Se l'espressione è vera, viene restituito `valore_se_vera`, altrimenti viene restituito `valore_se_falsa`.

```
#include <stdio.h>

int main(void)
{
    int x = 10;
    int y = (x > 5) ? 1 : 0;

    printf("y = %d\n", y);
}
```

Questo modo di scrivere l'istruzione `if-else` è utile quando si vuole assegnare un valore a una variabile in base a una condizione con una sola riga di codice. Tuttavia, se l'espressione è complessa, l'uso dell'if ternario può rendere il codice meno leggibile. In questi casi si sconsiglia di utilizzarlo e si preferisce l'istruzione `if-else` standard.

`switch`

Tanti if in cascata potrebbero essere sostituiti da uno switch. La sintassi è la seguente:

```
switch (espressione) {
    case constant-expression:
        // statements
        break;
    case constant-expression:
        // statements
        break;
    // ...
    default:
        // statements
}
```

L'espressione a fianco dell'istruzione `case` è senza variabili o chiamate a funzioni. L'istruzione `break` è necessaria per terminare il blocco di codice. Se non è presente, il controllo del flusso di esecuzione continua con il blocco di codice successivo. L'istruzione `default` è opzionale e viene eseguita se nessuna delle costanti corrisponde all'espressione.

Un esempio di utilizzo dello switch è il seguente:

```
#include <stdio.h>

int main() {
    int x = 2;

    switch (x) {
        case 1:
            printf("x è 1\n");
            break;
        case 2:
            printf("x è 2\n");
            break;
        default:
            printf("x non è nè 1 nè 2\n");
    }

    return 0;
}
```

> **!** Importante
>
> Lo `switch` può essere utilizzato solo con espressioni di tipo intero (`int`, `char`, `short`, `long`, ecc.) e con espressioni di tipo `enum` (che verrà trattato in seguito).

Cicli

Caratteristica fondamentale di un linguaggio di programmazione sono i cicli. C implementa i classici for, while, ma anche do…while

- `while (expression) { statements }`
- `do { statements } while (expression)`
- `for (initialization; condition; increment) { statements }`

Queste `while` e `for` sono equivalenti: tutto ciò che si può fare con un ciclo `for` si può fare con un ciclo `while` e viceversa. La scelta di quale istruzione utilizzare dipende dal contesto e dalla preferenza personale.

L'istruzione `do-while` è simile a `while`, ma la condizione viene valutata alla fine del blocco di codice. Questo significa che il blocco di codice viene eseguito almeno una volta, anche se la condizione è falsa.

> **i** Nota
>
> Dal C99 è possibile dichiarare variabili all'interno del ciclo `for`. Queste variabili sono visibili solo all'interno del ciclo. In C89 e nelle versioni precedenti, le variabili devono essere dichiarate all'inizio del blocco di codice.

```c
for (int i = 0; i < 10; i++) {
    // ...
}
```

Cicli infiniti

Un ciclo infinito è un ciclo che non termina mai. Un ciclo infinito può essere creato utilizzando un'istruzione `while` con una condizione sempre vera, ad esempio `while (1)`, oppure utilizzando un'istruzione `for` senza condizione, ad esempio `for (;;)`. Un ciclo infinito può essere interrotto con un'istruzione `break`.

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
```

```
    while (1)
    {
        printf("%d\n", i);
        i++;
        if (i == 10)
            break;
    }
}
```

`break` e `continue`

L'istruzione `break` termina immediatamente il ciclo in cui si trova. L'istruzione `continue` termina l'iterazione corrente del ciclo e passa alla successiva. Entrambe le istruzioni possono essere utilizzate all'interno di cicli `for`, `while` e `do-while`. L'istruzione `break` può essere utilizzata anche all'interno di un blocco `switch` per terminare immediatamente l'esecuzione del blocco.

> **Avviso**
>
> È bene non abusare di `break` e `continue`, perché possono rendere il codice meno leggibile. In generale, è meglio evitare di utilizzare `break` e 'continue all'interno di cicli annidati.

Array

Introduciamo ora gli array, la prima **struttura dati** che affronteremo.

Generalmente una variabile può contenere un solo valore, spesso però si vuole eseguire operazioni su una sequenza di variabili, una soluzione è l'utilizzo di un **array** (o vettore). Un array è una collezione di elementi **omogenei** (tutti dello stesso tipo).

La dichiarazione di un array avviene specificando il tipo e il numero degli elementi:

```
int a[10]
```

> **Consiglio**
>
> L'introduzione di costanti hardcoded produce codice difficile da rifattorizzare e mantenere, per questo, spesso, si stabilisce ad inizio programma una costante (o una macro) per scrivere un valore in un unico posto:
>
> ```
> #define N 10
> int a[N];
> ```

oppure:

```
const int N = 10;
int a[N];
```

nei precedenti esempi viene definita N ad inizio programma, nel resto del codice basterà usare questa costante quando ci si deve riferire alla dimensione dell'array a. Qualora ci si accorgesse in futuro che 10 non sia un numero sufficiente per contenere i dati del programma, basterà cambiare una sola riga di codice.

Si accede a un elemento dell'array tramite **subscripting**:

```
int a[2];
a[0] = 1;
a[1] = 2;
```

Gli indici degli array in C partono da 0, come in Java.

> **Nota**
>
> L'espressione `a[i]` (o simili) è un *lvalue* (left-value), ossia può essere usata come variabile, può quindi trovarsi a sinistra di un'assegnazione.

Il compilatore C non controlla i limiti degli array, quindi è possibile accedere a elementi fuori dal range definito, questo può causare errori difficili da individuare.

```
int a[10], i;
for (i = 0; i <= 10; i++) {
    a[i] = 0;
}
```

In certi compilatori il codice sopra riportato causa un loop infinito (più spesso però genera un errore a runtime e ferma l'esecuzione).

Si intuisce che C sia **molto permissivo** con il subscripting, il seguente codice quindi è perfettamente legittimo:

```
int a[50];
int i = 0, j = 3;
a[i+j*10] = 0;

i = 0;
while (i < j)
    a[i++] = 0;
```

Inizializzazione

Un array può essere inizializzato al momento della dichiarazione:

```
int a[5] = {1, 2, 3, 4, 5};
```

Se si mettono meno numeri, i restanti elementi vengono inizializzati a 0 (ma questo non succede se non ne specifico nessuno!):

```
int a[5] = {1, 2, 3}; // a = {1, 2, 3, 0, 0}
int b[4] = {0}; // b = {0, 0, 0, 0}
```

Se metto più elementi di quelli dichiarati il compilatore segnala un errore. Se invece si omette la dimensione dell'array, il compilatore la calcola automaticamente dalla lista di initializzazione:

```
int a[] = {1, 2, 3}; // a ha dimensione 3
```

Dimensione di un array

In C non esiste un sistema per ottenere senza sforzo la dimensione di un array, per questo, spesso, si utilizza una variabile per memorizzare la dimensione dell'array.

Tuttavia si può usare `sizeof` per ottenere la dimensione di un array:

```
int a[10];
sizeof(a); /* restituisce 40 su una macchina a 32 bit */
sizeof(a[0]); /* 4 */
sizeof(a)/sizeof(a[0]); /* 10, la dimensione dell'array */
```

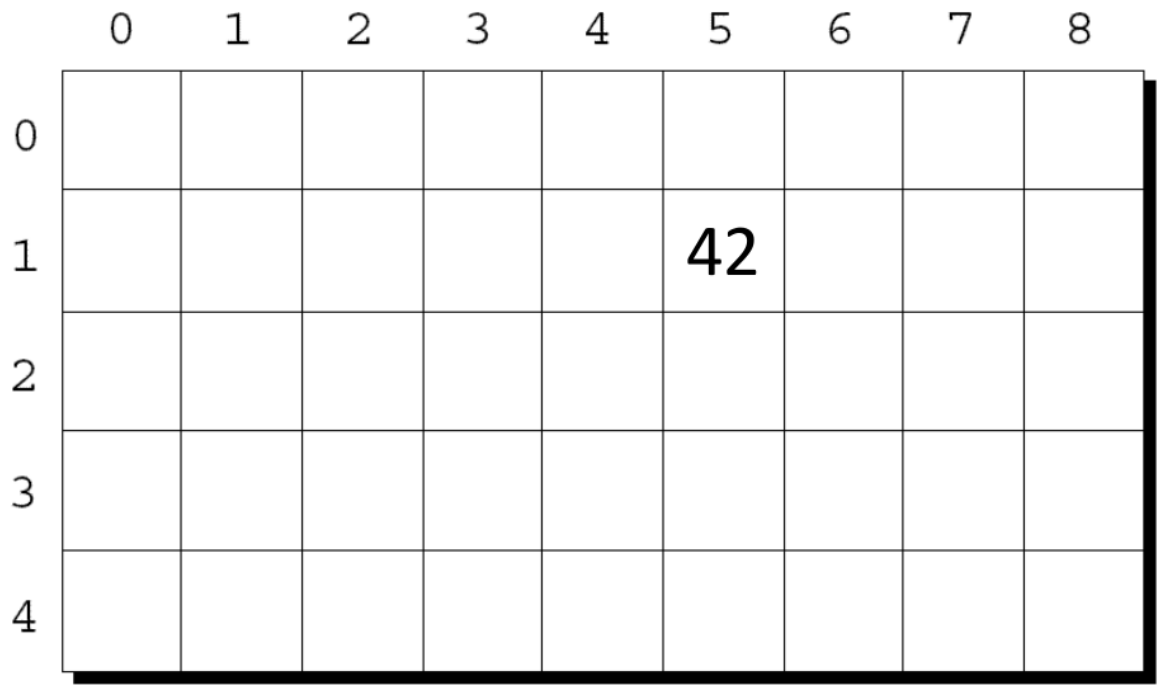
Array multidimensionali

Un array può contenere elementi di qualsiasi tipo, anche altri array! Per dichiarare un array multidimensionale si specifica il numero di elementi per ogni dimensione:

`int matrix[5][9];`

Per accedere a un elemento di un array multidimensionale si usano più indici:

`matrix[1][5] = 42;` e NON `matrix[1,5] = 42;`



Funzioni

Una funzione è una associazione tra due insiemi: il dominio e il codominio: $f : A \to B$

In programmazione il concetto di funzione è simile: una funzione è un blocco di codice che accetta un certo numero di argomenti e, spesso, restituisce un valore. Oltre al valore, però, una funzione può anche avere *side effects*, cioè modificare lo stato del programma.

In C si può pensare a una qualsiasi funzione come una scatola nera con un certo numero di ingressi e **un solo** output.

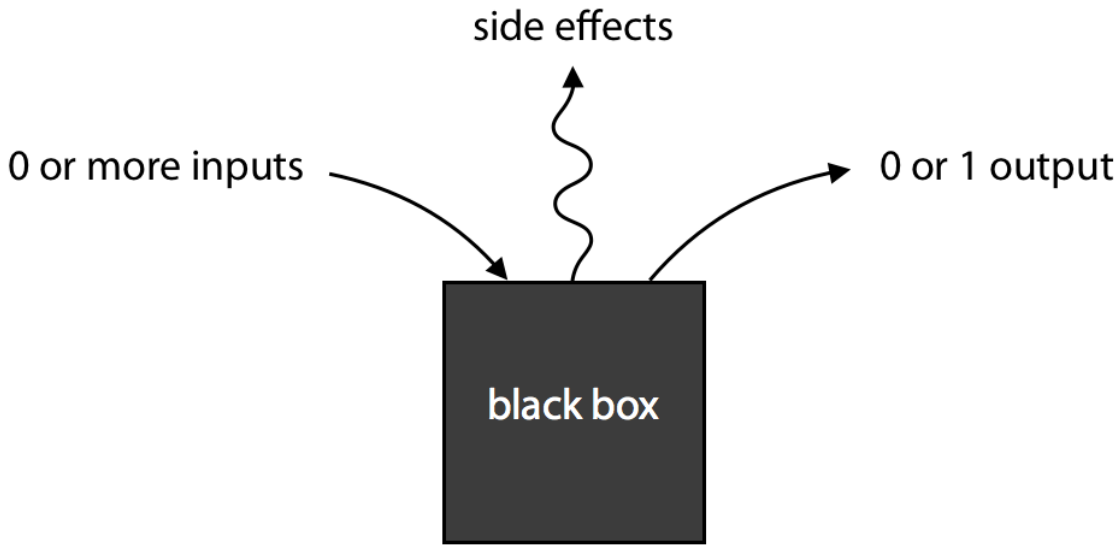


Figura 1: Funzione come una scatola nera.

Praticamente una funzione è un sottoprogramma, un insieme di istruzioni che esegue un compito specifico.

```
return_type function_name( parameters )
{
    declarations
    statements
}
```

Esempio:

```
double average(int a, int b)
{
    double average;
    average = (a + b) / 2.0;
    return average;
}
```

Il tipo di ritorno può essere `void`, in tal caso la funzione non restituisce alcun valore:

```
void print_int(int a)
{
    printf("%d\n", a);
    return;
}
```

> **Nota**
>
> L'istruzione `return` può essere omessa in una funzione `void`.

Le funzioni dichiarate precedentemente nel programma possono essere eseguite in blocchi di codice successivi.

Ogni chiamata di funzione è una *espressione*, e viene valutata con il suo valore di ritorno.

Riassunto

In questa lezione sono state introdotti molti elementi, di seguito un elenco che ripercorre i concetti affrontati:

- tipi di dato (int, float, char, double);
- espressioni booleane:
  - true e false sono in realtà 1 e 0;
  - operatori logici;
- operatori;
- control flow (if, switch, while, do-while, for);
- array:
  - inizializzazione degli array;
  - array multidimensionali;
- funzioni: sintassi e ricorsione;