Programmazione 1

Lezione 5

Dipartimento di Matematica *Federigo Enriques*
Università degli Studi di Milano

Gli array: introduzione

- In gergo informatico, un array è una collezione di dati dello stesso tipo, diciamo T, cui si può accedere tramite un indice, che è un valore intero non negativo.

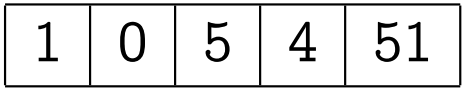
Gli array: introduzione

- In gergo informatico, un array è una collezione di dati dello stesso tipo, diciamo `T`, cui si può accedere tramite un indice, che è un valore intero non negativo.
- In italiano si usa il termine inglese array, oppure a volte il termine vettore. Si parla di *array di* `T` o di *vettore di* `T`.

Gli array: introduzione

- In gergo informatico, un array è una collezione di dati dello stesso tipo, diciamo T, cui si può accedere tramite un indice, che è un valore intero non negativo.
- In italiano si usa il termine inglese array, oppure a volte il termine vettore. Si parla di *array di* T o di *vettore di* T.
- La rappresentazione tipica di una array di T nella memoria centrale del computer è costituita da una successione di celle di memoria adiacenti, ciascuna atta a memorizzare un dato del tipo T. L'array ha una dimensione o lunghezza: il numero di elementi del tipo T che è in grado di memorizzare.

- Il modo tipico di raffigurarsi, ad esempio, un array di `int` di lunghezza 5 è il seguente:



- Il modo tipico di raffigurarsi, ad esempio, un array di `int` di lunghezza 5 è il seguente:

1	0	5	4	51

- Le celle sono numerate consecutivamente a partire da *zero*:

Indice	0	1	2	3	4
Array	1	0	5	4	51

- Il modo tipico di raffigurarsi, ad esempio, un array di `int` di lunghezza 5 è il seguente:

1	0	5	4	51

- Le celle sono numerate consecutivamente a partire da *zero*:

Indice	0	1	2	3	4
Array	1	0	5	4	51

- Se il nome di questo array è `pippo`, allora per accedere all'elemento i-esimo si usa la notazione `pippo[i]`. Per esempio,

  `pippo[3]`

  è un'espressione di tipo `int` e valore 4. Invece,

  `pippo[0]`

  è un'espressione di tipo `int` e valore 1.

- Nota Bene 1. In C la numerazione degli elementi degli array *comincia da* 0. Nell'esempio, l'array `pippo` ha dimensione 5 — e quindi consta di 5 elementi in tutto — il suo *primo* elemento è `pippo[0]`, e il suo *ultimo* elemento è `pippo[4]`.

- Nota Bene 1. In C la numerazione degli elementi degli array *comincia da* 0. Nell'esempio, l'array `pippo` ha dimensione 5 — e quindi consta di 5 elementi in tutto — il suo *primo* elemento è `pippo[0]`, e il suo *ultimo* elemento è `pippo[4]`.
- Nota Bene 2. Nell'esempio, l'espressione

  `pippo[6]`

  è *sintatticamente corretta*, ma il suo uso costituirebbe un *errore di programmazione*: la posizione numero 6 dell'array `pippo` non esiste.

- Nota Bene 1. In C la numerazione degli elementi degli array *comincia da* 0. Nell'esempio, l'array `pippo` ha dimensione 5 — e quindi consta di 5 elementi in tutto — il suo *primo* elemento è `pippo[0]`, e il suo *ultimo* elemento è `pippo[4]`.
- Nota Bene 2. Nell'esempio, l'espressione

  `pippo[6]`

  è *sintatticamente corretta*, ma il suo uso costituirebbe un *errore di programmazione*: la posizione numero 6 dell'array `pippo` non esiste. L'esecuzione di una tale istruzione comporta l'accesso a una zona della memoria centrale non assegnata al programma (o "non allocata", come si dice in gergo), circostanza che segnala sempre un *errore grave*, e che porta spesso a un'interruzione anomala dell'esecuzione del programma con errore di tipo segmentation fault.

- Sintassi. Per dichiarare un array di `int` di nome `pippo` e dimensione `5` si usa l'istruzione:

```
int pippo[5];
```

- Sintassi. Per dichiarare un array di `int` di nome `pippo` e dimensione `5` si usa l'istruzione:

```
int pippo[5];
```

- Sintassi. Più in generale, per dichiarare un array di `T` di nome `nome` e dimensione `n` si usa l'istruzione:

```
T nome[n];
```
($\star$)

- Sintassi. Per dichiarare un array di `int` di nome `pippo` e dimensione `5` si usa l'istruzione:

```
int pippo[5];
```

- Sintassi. Più in generale, per dichiarare un array di `T` di nome `nome` e dimensione `n` si usa l'istruzione:

```
T nome[n];
```
$(\star)$

- Semantica (allocazione). L'esecuzione di $(\star)$ alloca al programma n posizioni di memoria consecutive (nella RAM, la memoria centrale del calcolatore), numerate da 0 a $n-1$, ciascuna atta a contenere un dato di tipo `T`.

- Sintassi. Per dichiarare un array di `int` di nome `pippo` e dimensione `5` si usa l'istruzione:

```
int pippo[5];
```

- Sintassi. Più in generale, per dichiarare un array di `T` di nome `nome` e dimensione `n` si usa l'istruzione:

```
T nome[n];
```
$(\star)$

- Semantica (allocazione). L'esecuzione di $(\star)$ alloca al programma n posizioni di memoria consecutive (nella RAM, la memoria centrale del calcolatore), numerate da 0 a $n-1$, ciascuna atta a contenere un dato di tipo `T`.
- Nota Bene 3. Dopo l'esecuzione di $(\star)$ le posizioni di memoria allocate *non* sono inizializzate ad un valore predeterminato: il loro valore è dunque da considerarsi indeterminato.

array1.c

```c
#include <stdio.h>

int main(void)
{
        //dichiara array di 10 int
        //da v[0] a v[9]
        int v[10];

        for(int i=0; i<10; i++)
                //stampa elemento i-esimo,
                //non inizializzato
                printf("%d\n",v[i]);

        return 0;
}
```

array2.c

```c
#include <stdio.h>

int main(void)
{
    //dichiara array di 10 int
    //da v[0] a v[9]
    int v[10];
    //variabile indice
    int i;

    for(i=0; i<10; i++)
        v[i]=0;                 //iniz. elemento i-esimo a 0

    for(i=0; i<10; i++)
        printf("%d\n",v[i]); //stampa elemento i-esimo

    return 0;
}
```

array3.c

```c
#include <stdio.h>

int main(void)
{
        //dichiara array di 10 int
        //da v[0] a v[9]
        int v[10];

        //ERRORE: accede a elemento inesistente
        printf("%d\n",v[10]);

        return 0;
}
```

array4.c

```c
#include <stdio.h>

int main(void)
{
    int v[10];
    int w[10];
    //variabile indice
    int i;

    for(i=0; i<10; i++)
        v[i]=i*i;    //iniz. elemento i-esimo a i^2

    for(i=0; i<10; i++)
        w[i]=v[9-i]; //w e' v in ordine inverso

    for(i=0; i<10; i++)
        printf("%d\t%d\n",v[i],w[i]);

    return 0;
}
```

Esercizio in classe: `Istogrammi orizzontali`

Esercizio.

Scrivere un programma che chieda all'utente di inserire una successione finita di valori interi non negativi, e visualizzi un istogramma (orizzontale) di asterischi $*$ che rappresenti i valori inseriti.

Esempio. Se i valori in ingresso fossero 10 e fossero, nell'ordine, $2, 1, 5, 4, 4, 4, 6, 1, 7, 4$, il programma dovrebbe visualizzare:

```
**
*
*****
****
****
****
******
*
*******
****
```

isto.c

```c
#include <stdio.h>
int main(void)
{
    int i,n;
    printf("Quanti interi vorrai inserire? ");    //lettura 1
    scanf("%d%*c",&n);
    int v[n];
    for(i=0; i<n; i++)                            //lettura 2
        do
        {
            printf("Inserisci intero non negativo %d-esimo: ",i+1);
            scanf("%d%*c",&(v[i]));
        }
        while(v[i] < 0);
    for(i=0; i<n; i++)                            //scrittura
    {
        for(int j=0;j<v[i];j++)
        printf("*");
        printf("\n");
    }
    return 0;
}
```

Le stringhe in C: array di char

- In gergo informatico, una stringa è una successione (finita) di caratteri. Molti linguaggi di programmazione mettono a disposizione un tipo stringa apposito.

Le stringhe in C: array di char

- In gergo informatico, una stringa è una successione (finita) di caratteri. Molti linguaggi di programmazione mettono a disposizione un tipo stringa apposito.
- In C, invece, si usano gli array di char per implementare le stringhe.

Le stringhe in C: array di char

- In gergo informatico, una stringa è una successione (finita) di caratteri. Molti linguaggi di programmazione mettono a disposizione un tipo stringa apposito.
- In C, invece, si usano gli array di char per implementare le stringhe.
- Per convenzione, in C le stringhe (=array di char) sono terminati dal carattere speciale \0, il cui codice ASCII è zero. Non si confonda \0 con \n.

Le stringhe in C: array di char

- In gergo informatico, una stringa è una successione (finita) di caratteri. Molti linguaggi di programmazione mettono a disposizione un tipo stringa apposito.
- In C, invece, si usano gli array di `char` per implementare le stringhe.
- Per convenzione, in C le stringhe (=array di `char`) sono terminati dal carattere speciale `\0`, il cui codice ASCII è zero. Non si confonda `\0` con `\n`.
- Quindi, la stringa

  `Ciao`

  è implementata in C tramite l'array di `char` di dimensione 5:

Indice	0	1	2	3	4
Array	`'C'`	`'i'`	`'a'`	`'o'`	`'\0'`

- La convenzione sul carattere di terminazione è utile quando è necessario eseguire elaborazioni sulle stringhe. Per esempio, si può usare il carattere di conversione %s assieme a `printf` per visualizzare un array di `char` *fino al primo carattere* `\0` che `printf` incontra in memoria.

- La convenzione sul carattere di terminazione è utile quando è necessario eseguire elaborazioni sulle stringhe. Per esempio, si può usare il carattere di conversione %s assieme a `printf` per visualizzare un array di `char` *fino al primo carattere* `\0` che `printf` incontra in memoria.
- Come altro esempio, supponiamo di leggere una stringa dal terminale. Non sappiamo quanto sarà lunga la stringa. Allora predisporremo un array di `char` grande a sufficienza da contenere i dati. Per esempio:

```
char input[1024]
```

- La convenzione sul carattere di terminazione è utile quando è necessario eseguire elaborazioni sulle stringhe. Per esempio, si può usare il carattere di conversione %s assieme a `printf` per visualizzare un array di `char` *fino al primo carattere* `\0` che `printf` incontra in memoria.
- Come altro esempio, supponiamo di leggere una stringa dal terminale. Non sappiamo quanto sarà lunga la stringa. Allora predisporremo un array di `char` grande a sufficienza da contenere i dati. Per esempio:

```
char input[1024]
```

- Se l'utente inserisce meno di 1024 caratteri, possiamo indicare il punto in cui termina la stringa inserita tramite `\0`.

- La convenzione sul carattere di terminazione è utile quando è necessario eseguire elaborazioni sulle stringhe. Per esempio, si può usare il carattere di conversione %s assieme a `printf` per visualizzare un array di `char` *fino al primo carattere* `\0` che `printf` incontra in memoria.
- Come altro esempio, supponiamo di leggere una stringa dal terminale. Non sappiamo quanto sarà lunga la stringa. Allora predisporremo un array di `char` grande a sufficienza da contenere i dati. Per esempio:

```
char input[1024]
```

- Se l'utente inserisce meno di 1024 caratteri, possiamo indicare il punto in cui termina la stringa inserita tramite `\0`.
- Il codice che esegue algoritmi sulla stringa memorizzata nell'array `input` potrà usare il carattere `\0` per sapere dove finisce effettivamente la stringa.

Lettura delle stringhe da terminale

Per esercitarvi in laboratorio, dovrete essere in grado di eseguire la lettura di una stringa di caratteri digitata dall'utente sul terminale. Vedremo adesso come fare, *senza scendere nei dettagli*. Approfondiremo l'argomento in seguito.

Lettura delle stringhe da terminale

Per esercitarvi in laboratorio, dovrete essere in grado di eseguire la lettura di una stringa di caratteri digitata dall'utente sul terminale. Vedremo adesso come fare, *senza scendere nei dettagli*. Approfondiremo l'argomento in seguito.

Un primo modo per leggere una stringa è di usare la funzione `scanf` di `stdio.h`, che già conosciamo, con carattere di conversione `%s`.

Lettura delle stringhe da terminale

Per esercitarvi in laboratorio, dovrete essere in grado di eseguire la lettura di una stringa di caratteri digitata dall'utente sul terminale. Vedremo adesso come fare, *senza scendere nei dettagli*. Approfondiremo l'argomento in seguito.

Un primo modo per leggere una stringa è di usare la funzione `scanf` di `stdio.h`, che già conosciamo, con carattere di conversione `%s`.

Un secondo modo, solitamente preferibile, è di usare la funzione `fgets` di `stdio.h`.

Lettura delle stringhe da terminale

Per esercitarvi in laboratorio, dovrete essere in grado di eseguire la lettura di una stringa di caratteri digitata dall'utente sul terminale. Vedremo adesso come fare, *senza scendere nei dettagli*. Approfondiremo l'argomento in seguito.

Un primo modo per leggere una stringa è di usare la funzione `scanf` di `stdio.h`, che già conosciamo, con carattere di conversione `%s`.

Un secondo modo, solitamente preferibile, è di usare la funzione `fgets` di `stdio.h`.

In entrambi i casi dovremo predisporre un array di `char` di grandezza sufficiente a memorizzare la stringa digitata dall'utente. Per esempio, si può dimensionare questo array usando la costante simbolica `BUFSIZ` (per *buffer size*) definita in `stdio.h`.

Lettura delle stringhe da terminale

I comportamenti di `scanf` e di `fgets` nella lettura di una stringa sono diversi.

- `scanf` legge la successione di caratteri proveniente dal terminale fino al primo carattere di spaziatura, di tabulazione (`\t`) o di ritorno a capo (`\n`), *senza* includere tale carattere nella stringa memorizzata.
- `fgets` legge la successione di caratteri proveniente dal terminale fino al primo carattere di ritorno a capo (`\n`), *includendo* tale carattere nella stringa memorizzata. (Dunque, `fgets` include anche spazi e tabulazioni nella stringa letta.)
- Entrambe le funzioni aggiungono in coda alla successione di caratteri letta dal terminale il carattere `\0`.

Lettura delle stringhe da terminale

Supponiamo di aver dichiarato un array di `char` in questo modo:

```
char str[BUFSIZ];
```

L'istruzione

```
scanf("%s",str); // *non* &str
```

memorizza la stringa letta dal terminale in `str`. L'istruzione

```
fgets(str,BUFSIZ,stdin);
```

esegue la stessa operazione, con i distinguo visti alla diapositiva precedente. Il parametro `stdin` indica che la lettura avverrà da terminale e non, per esempio, da un file che risieda su memoria di massa. In assenza di altre indicazioni o di motivazioni ben chiare, *usate* `fgets` *e non* `scanf` *per leggere stringhe.*

Esercizio in classe: `Eco speculare`

Esercizio.

Scrivere un programma che chieda all'utente di inserire una stringa, e visualizzi la stringa in ordine inverso.

Esempio. Se la stringa in ingresso fosse

```
Ciao
```

la stringa in uscita dovrebbe essere

```
oaiC
```

rev.c

```c
#include <stdio.h>
int main(void)
{
    char v[256]; //conterra' la stringa letta
    printf("Scrivi qualcosa: ");
    fgets(v,256,stdin);
    /* v adesso contiene la stringa letta, incluso '\n',
       e terminata da '\0'. Calcolo la lunghezza della
       stringa inserita dall'utente. */
    int l=0;
    for (l=0; v[l]!='\0'; l++);

    //visualizza v in ordine inverso, omettendo il '\n' finale
    l=l-2; //v[l]=='\0'. v[l-1]=='\n'.
    for (;l>=0;l--)
        printf("%c",v[l]);
    printf("\n");
    return 0;
}
```

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.
2. Output: Uso di `printf`. Input: Primi usi di `getchar`, `scanf`, `fgets`.

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.
2. Output: Uso di `printf`. Input: Primi usi di `getchar`, `scanf`, `fgets`.
3. Operatori aritmetici, logici, relazionali:

   `+`, `-`, `*`, `/`, `%`, `!`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, `>=`.

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.
2. Output: Uso di `printf`. Input: Primi usi di `getchar`, `scanf`, `fgets`.
3. Operatori aritmetici, logici, relazionali:

   `+`, `-`, `*`, `/`, `%`, `!`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, `>=`.

4. Operatori di pre- e post-incremento, operatori di assegnamento derivati:

   `++`, `-`, `+=`, `-=`, `*=`, `/=`, `%=`

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.
2. Output: Uso di `printf`. Input: Primi usi di `getchar`, `scanf`, `fgets`.
3. Operatori aritmetici, logici, relazionali:

   `+`, `-`, `*`, `/`, `%`, `!`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, `>=`.

4. Operatori di pre- e post-incremento, operatori di assegnamento derivati:

   `++`, `-`, `+=`, `-=`, `*=`, `/=`, `%=`

5. Flusso del controllo e programmazione strutturata:
   - *Sequenza*: $istruzione_1 \ldots istruzione_n$
   - *Selezione*: `if-else`, `else-if`, `switch`.
   - *Iterazione*: `while`, `do-while`, `for`.

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.
2. Output: Uso di `printf`. Input: Primi usi di `getchar`, `scanf`, `fgets`.
3. Operatori aritmetici, logici, relazionali:

   `+`, `-`, `*`, `/`, `%`, `!`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, `>=`.

4. Operatori di pre- e post-incremento, operatori di assegnamento derivati:

   `++`, `-`, `+=`, `-=`, `*=`, `/=`, `%=`

5. Flusso del controllo e programmazione strutturata:
   - *Sequenza*: $istruzione_1 \ldots istruzione_n$
   - *Selezione*: `if-else`, `else-if`, `switch`.
   - *Iterazione*: `while`, `do-while`, `for`.
6. Primi usi degli array.

Le funzioni nella programmazione, in generale

- Astrattamente un programma computa una funzione che associa dati in uscita (l'output) a dati in ingresso (l'input).

Le funzioni nella programmazione, in generale

- Astrattamente un programma computa una funzione che associa dati in uscita (l'output) a dati in ingresso (l'input).
- Così come, in matematica, le funzioni si possono comporre, allo stesso modo i programmi possono richiamare, o invocare, altre funzioni ausiliarie al fine di eseguire la computazione richiesta.

Le funzioni nella programmazione, in generale

- Astrattamente un programma computa una funzione che associa dati in uscita (l'output) a dati in ingresso (l'input).
- Così come, in matematica, le funzioni si possono comporre, allo stesso modo i programmi possono richiamare, o invocare, altre funzioni ausiliarie al fine di eseguire la computazione richiesta.
- Esempio. Data $f : \mathbb{Z} \to \mathbb{Z}$ definita da $n \in \mathbb{Z} \mapsto n^2 + 3 \in \mathbb{Z}$, si ha:

$$f(n) = (t \circ q)(n) = t(q(n)), \text{ dove}$$
$$t(n) := n + 3$$
$$q(n) := n^2.$$

Le funzioni nella programmazione, in generale

- Il programma corrispondente alla composizione $t \circ q$ avrebbe questa struttura:

struttura1.c

```
1  int f(int n)
2  {
3      int x = q(n);
4      int y = t(x);
5      return y;
6  }
```

- Bisogna però implementare anche `t` e `q`, che sono a loro volta funzioni. E bisogna anche aggiungere la funzione iniziale `main`. Ecco il programma completo.

Le funzioni nella programmazione, in generale

struttura2.c

```c
#include <stdio.h>

int q(int n)
{
    return n*n;
}
int t(int n)
{
    return n+3;
}
int f(int n)
{
    int x = q(n);
    int y = t(x);
    return y;
}
int main(void)
{
    printf("%d\n",f(2)); return 0;
}
```

Le funzioni nella programmazione, in generale

- Nella programmazione, le funzioni — a volte anche dette procedure o subroutine — costituiscono un comodo strumento per incapsulare una data computazione che abbia senso compiuto in sé stessa.

Le funzioni nella programmazione, in generale

- Nella programmazione, le funzioni — a volte anche dette procedure o subroutine — costituiscono un comodo strumento per incapsulare una data computazione che abbia senso compiuto in sé stessa.
- Una volta scritto il codice di una funzione, per poterla usare non è più necessario preoccuparsi dei dettagli della sua implementazione: basterà sapere *cosa* la funzione fa, ignorando i dettagli di *come* lo faccia. Una buona organizzazione del codice in unità funzionali indipendenti è un ingrediente importante della buona programmazione.

Le funzioni nella programmazione, in generale

- Nella programmazione, le funzioni — a volte anche dette procedure o subroutine — costituiscono un comodo strumento per incapsulare una data computazione che abbia senso compiuto in sé stessa.
- Una volta scritto il codice di una funzione, per poterla usare non è più necessario preoccuparsi dei dettagli della sua implementazione: basterà sapere *cosa* la funzione fa, ignorando i dettagli di *come* lo faccia. Una buona organizzazione del codice in unità funzionali indipendenti è un ingrediente importante della buona programmazione.
- Per programmare bene in C è essenziale usare in modo competente le funzioni: il linguaggio conduce in modo naturale alla stesura di programmi costituiti da molte funzioni, tipicamente brevi.

Le funzioni nella programmazione C

- Sintassi.

```
tipo nome(tipo_1 param_1,...,tipo_n param_n)
{
  ⋮     (implementazione)
}
```

Le funzioni nella programmazione C

- Sintassi.

```
tipo nome(tipo_1 param_1,...,tipo_n param_n)
{
  ⋮    (implementazione)
}
```

- La riga

```
tipo nome(tipo_1 param_1,...,tipo_n param_n);
```

si chiama prototipo della funzione. Si noti che il prototipo, terminato da `';'`, è *privo* del corpo, che in effetti costituisce la sua implementazione.

Le funzioni nella programmazione C

- Sintassi.

```
tipo nome(tipo_1 param_1,...,tipo_n param_n)
{
  ⋮     (implementazione)
}
```

- La riga

```
tipo nome(tipo_1 param_1,...,tipo_n param_n);
```

  si chiama prototipo della funzione. Si noti che il prototipo, terminato da `';'`, è *privo* del corpo, che in effetti costituisce la sua implementazione.

- Casi particolari. `void nome(void)` — nessun valore in ingresso, nessun valore restituito.

Le funzioni nella programmazione C

- Sintassi.

```
tipo nome(tipo_1 param_1,...,tipo_n param_n)
{
  ⋮    (implementazione)
}
```

- La riga

```
tipo nome(tipo_1 param_1,...,tipo_n param_n);
```

si chiama prototipo della funzione. Si noti che il prototipo, terminato da `';'`, è *privo* del corpo, che in effetti costituisce la sua implementazione.

- Casi particolari. `void nome(tipo_1 param_1,...,tipo_n param_n)` — valori in ingresso, nessun valore restituito.

Le funzioni nella programmazione C

- Sintassi.

```
tipo nome(tipo_1 param_1,...,tipo_n param_n)
{
  ⋮     (implementazione)
}
```

- La riga

```
tipo nome(tipo_1 param_1,...,tipo_n param_n);
```

  si chiama prototipo della funzione. Si noti che il prototipo, terminato da `';'`, è *privo* del corpo, che in effetti costituisce la sua implementazione.

- Casi particolari. `tipo nome(void)` — nessun valore in ingresso, un valore restituito.

Le funzioni nella programmazione C

- Sintassi.

```
tipo nome(tipo_1 param_1,...,tipo_n param_n)
{
  ⋮     (implementazione)
}
```

- La riga

```
tipo nome(tipo_1 param_1,...,tipo_n param_n);
```

  si chiama prototipo della funzione. Si noti che il prototipo, terminato da `';'`, è *privo* del corpo, che in effetti costituisce la sua implementazione.

- `void` è una parola chiave del C che segnala quindi l'assenza di parametri.

funz1.c

```c
#include <stdio.h>

void stampa(void) {
    printf("Ciao Mondo.\n");
    return;
}

int main(void) {
    stampa();
    return 0;
}
```

La parola chiave `void` si può omettere quando è usata per indicare il fatto che una funzione non ha argomenti in ingresso, come avviene nelle diapositive seguenti per la funzione `stampa`.

funz2.c

```c
#include <stdio.h>

int main(void) {
    stampa(); //Implicitamente dichiarata int
    return 0;
}

void stampa() { //Esplicitamente dichiarata void
    printf("Ciao Mondo.\n");
    return;
}
```

La parola chiave `void`, però, non va *mai* omessa quando è usata per indicare il fatto che una funzione non restituisce valori: infatti, in sua assenza il compilatore assume che il tipo restituito dalla funzione sia `int`. Ciò avviene per garantire compatibilità con le versioni storiche del C, ma può condurre ad incoerenze ed errori. Nell'esempio, la compilazione *riscontra un errore*.

funz3.c

```c
#include <stdio.h>

int main(void) {
    /* Dichiarazione di funzione */
    void stampa();

    stampa();
    return 0;
}

/* Definizione (=dichiarazione+implementazione) di funzione */
void stampa() {
    printf("Ciao Mondo.\n");
    return;
}
```

Si noti con cura la differenza fra dichiarazione e definizione della funzione: la prima ne stabilisce il *prototipo*, la seconda *sia il prototipo che l'implementazione*.

funz3.c

```
1  #include <stdio.h>
2
3  int main(void) {
4      /* Dichiarazione di funzione */
5      void stampa();
6
7      stampa();
8      return 0;
9  }
10
11 /* Definizione (=dichiarazione+implementazione) di funzione */
12 void stampa() {
13     printf("Ciao Mondo.\n");
14     return;
15 }
```

La dichiarazione di una funzione serve a indicare al compilatore la parte di codice dove si intende utilizzare la funzione. In questo esempio, alla riga 5 dichiariamo di voler utilizzare `stampa` all'interno di `main`.

funz3.c

```c
#include <stdio.h>

int main(void) {
    /* Dichiarazione di funzione */
    void stampa();

    stampa();
    return 0;
}

/* Definizione (=dichiarazione+implementazione) di funzione */
void stampa() {
    printf("Ciao Mondo.\n");
    return;
}
```

Ne segue che una funzione è spesso *dichiarata più volte* in un programma, ed è quindi necessario che le diverse dichiarazioni della stessa funzione siano coerenti, a pena di un errore in compilazione.

funz3.c

```
1  #include <stdio.h>
2
3  int main(void) {
4      /* Dichiarazione di funzione */
5      void stampa();
6
7      stampa();
8      return 0;
9  }
10
11 /* Definizione (=dichiarazione+implementazione) di funzione */
12 void stampa() {
13     printf("Ciao Mondo.\n");
14     return;
15 }
```

D'altro canto, ogni funzione deve avere *esattamente una* implementazione in tutto il programma: in questo esempio, l'implementazione di stampa si trova a partire dalla riga 12.

funz4.c

```c
#include <stdio.h>

/* Dichiarazione esterna di funzione */
void stampa();

int main(void) {
    stampa();
    return 0;
}

/* Definizione (=dichiarazione+implementazione) di funzione */
void stampa() {
    printf("Ciao Mondo.\n");
    return;
}
```

Una funzione si può anche dichiarare al di fuori dell'implementazione di qualunque altra funzione, come in questo esempio. Si parla allora di una dichiarazione globale: il compilatore rende la funzione disponibile per l'uso a tutto il codice che segue il punto della dichiarazione.

par1.c

```c
#include <stdio.h>

int main(void) {
    void stampa(int);

    stampa(3);
    return 0;
}

void stampa(int n) {
    for (; n>0; n--)
        printf("Ciao Mondo (%d).\n",n);
    return;
}
```

Esempio di passaggio di un parametro (di tipo `int`).
Supponiamo ora di voler modificare l'esempio in modo che `stampa` si rifiuti di iterare la stampa per più di 100 volte.

par2.c

```c
#include <stdio.h>

int main(void) {
    int stampa(int);

    if ( stampa(3) < 0 )
        printf("Errore: il valore del parametro e' fuori misura.\n");
    return 0;
}

int stampa(int n) {          //precondizione: n <= 100
    if (n>100)
        return -1;          //segnala errore alla funzione chiamante
    for (; n>0; n--)
        printf("Ciao Mondo (%d).\n",n);
    return 0;               //segnala ok alla funzione chiamante
}
```

Passaggio di un parametro (di tipo `int`) e restituzione di un valore (di tipo `int`).

par2.c

```c
#include <stdio.h>

int main(void) {
    int stampa(int);

    if ( stampa(3) < 0 )
        printf("Errore: il valore del parametro e' fuori misura.\n");
    return 0;
}

int stampa(int n) {          //precondizione: n <= 100
    if (n>100)
        return -1;          //segnala errore alla funzione chiamante
    for (; n>0; n--)
        printf("Ciao Mondo (%d).\n",n);
    return 0;               //segnala ok alla funzione chiamante
}
```

Modifichiamo l'esempio: l'utente indicherà il numero di iterazioni.

Usiamo la funzione della libreria `atoi`, contenuta in `stdlib.h`.

par3.c

```c
#include <stdio.h>
#include <stdlib.h> //per atoi, che converte una stringa in intero
int main(void) {
    int stampa(int); char s[256];//stringa per lettura da console

    printf("Inserisci numero di stampe: ");
    fgets(s,256,stdin);
    if ( stampa(atoi(s)) < 0 )    //composizione di funzioni
        printf("Errore.\n");
    return 0;
}
int stampa(int n) {                 //precondizione: n <= 100
    if (n>100)
        return -1;                  //segnala errore alla funz. chiamante
    for (; n>0; n--)
        printf("Ciao Mondo (%d).\n",n);
    return 0;                       //segnala ok alla funz. chiamante
}
```

Esempio di composizione delle chiamate alle funzioni.

varglob.c

```c
#include <stdio.h>

int n; /* Variabile globale */

int main(void) {

        void funz(void);

        n=5;
        funz();
        printf("%d\n",n);
        return 0;
}

void funz(void) {
        n++;
        /* Posso omettere return se la funzione ha tipo void */
}
```

Anche le variabili, come le funzioni, possono essere dichiarate come variabili globali, al di fuori di qualunque funzione. In questo esempio, n è dichiarata globale alla riga 3, e risulta quindi accessibile sia da `main` che da `funz`. Cosa stamperà il programma, 5 oppure 6? Perché? Per approfondire, occorre studiare le cosiddette regole di visibilità, cosa che faremo adesso.

Il campo di visibilità

Il campo di visibilità — in inglese *scope* — di un identificatore (ossia, di un nome di variabile o di funzione) è la parte di programma in cui quell'identificatore può essere usato, in quanto è lì riconosciuto dal compilatore come definito.

Il campo di visibilità

Il campo di visibilità — in inglese *scope* — di un identificatore (ossia, di un nome di variabile o di funzione) è la parte di programma in cui quell'identificatore può essere usato, in quanto è lì riconosciuto dal compilatore come definito.
Lo standard C definisce il campo di visibilità di variabili e funzioni, stabilendo delle *regole di visibilità*. Queste regole variano a seconda dell'oggetto considerato.
In particolare, occorre distinguere:

- Variabili automatiche (e quindi locali a un blocco).
- Variabili esterne (e quindi globali a uno o più file sorgente).
- Funzioni.

Variabili automatiche (o locali)

- Il campo di visibilità di una variabile automatica si estende dal punto in cui essa è dichiarata — che necessariamente si trova all'interno di un blocco — fino alla fine del blocco.

Variabili automatiche (o locali)

- Il campo di visibilità di una variabile automatica si estende dal punto in cui essa è dichiarata — che necessariamente si trova all'interno di un blocco — fino alla fine del blocco.
- Variabili locali dallo stesso nome dichiarate in blocchi diversi, e non contenuti l'uno nell'altro, non hanno tra loro alcun rapporto.

Variabili automatiche (o locali)

- Il campo di visibilità di una variabile automatica si estende dal punto in cui essa è dichiarata — che necessariamente si trova all'interno di un blocco — fino alla fine del blocco.
- Variabili locali dallo stesso nome dichiarate in blocchi diversi, e non contenuti l'uno nell'altro, non hanno tra loro alcun rapporto.
- Lo stesso vale per i parametri delle funzioni, che sono a tutti gli effetti *variabili automatiche locali alla funzione*.

Variabili automatiche (o locali)

Nel caso però in cui due variabili automatiche *dallo stesso nome* siano dichiarate in due blocchi *di cui uno sia dentro l'altro*, occorre stabilire una regola per rendere non ambigui i riferimenti all'unico nome delle due variabili.

Variabili automatiche (o locali)

Nel caso però in cui due variabili automatiche *dallo stesso nome* siano dichiarate in due blocchi *di cui uno sia dentro l'altro*, occorre stabilire una regola per rendere non ambigui i riferimenti all'unico nome delle due variabili.

La regola è che la variabile dichiarata più in profondità fa ombra a, oppure oscura, quella dichiarata meno in profondità. (In inglese si parla di *shadowing* della variabile più interna su quella meno interna.)

Variabili automatiche (o locali)

Nel caso però in cui due variabili automatiche *dallo stesso nome* siano dichiarate in due blocchi *di cui uno sia dentro l'altro*, occorre stabilire una regola per rendere non ambigui i riferimenti all'unico nome delle due variabili.

La regola è che la variabile dichiarata più in profondità fa ombra a, oppure oscura, quella dichiarata meno in profondità. (In inglese si parla di *shadowing* della variabile più interna su quella meno interna.)

Ciò significa che le occorrenze del nome nel blocco più interno saranno riferite dal compilatore all variabile locale a quel blocco.

ombra.c

```c
#include <stdio.h>

int main(void)
{
    int x=5; //x locale al main
    printf("Main:\tx=%d\n",x);

    for (int i=0;i<3;i++) //i locale a tutto il for
    {
        int x=0; //x locale al *corpo* del for: fa ombra ad x in main
        x--;
        printf("For %d:\tx=%d\n",i,x);
    }

    printf("Main:\tx=%d\n",x);

    return 0;
}
```

Il nome `x` più interno (nel `for`) oscura quello più esterno (in `main`).

Variabili esterne (o globali)

- Il campo di visibilità di una variabile globale — cioè dichiarata in un file sorgente all'esterno di qualunque funzione contenuta in quel file — si estende dal punto in cui essa è dichiarata fino alla fine del file sorgente.

Variabili esterne (o globali)

- Il campo di visibilità di una variabile globale — cioè dichiarata in un file sorgente all'esterno di qualunque funzione contenuta in quel file — si estende dal punto in cui essa è dichiarata fino alla fine del file sorgente.
- Nel caso in cui sia necessario riferirsi alla variabile globale *prima* della sua dichiarazione, oppure in un altro file sorgente, occorre usare la parola chiave `extern`.

Variabili esterne (o globali)

- Il campo di visibilità di una variabile globale — cioè dichiarata in un file sorgente all'esterno di qualunque funzione contenuta in quel file — si estende dal punto in cui essa è dichiarata fino alla fine del file sorgente.
- Nel caso in cui sia necessario riferirsi alla variabile globale *prima* della sua dichiarazione, oppure in un altro file sorgente, occorre usare la parola chiave `extern`.
- Per spiegare il significato di `extern` distinguiamo fra dichiarazione e definizione di una variabile esterna.

Variabili esterne (o globali)

- La dichiarazione di una variabile globale ne stabilisce il nome e il tipo ma *non alloca spazio in memoria* per i suoi valori. Una tale dichiarazione globale è premessa dalla parola chiave `extern`. (Cfr. la dichiarazione di una funzione.)

Variabili esterne (o globali)

- La dichiarazione di una variabile globale ne stabilisce il nome e il tipo ma *non alloca spazio in memoria* per i suoi valori. Una tale dichiarazione globale è premessa dalla parola chiave `extern`. (Cfr. la dichiarazione di una funzione.)
- La definizione di una variabile globale, invece, omette `extern` e comporta l'allocazione della necessaria quantità di memoria. (Cfr. la definizione di una funzione.)

Variabili esterne (o globali)

- La dichiarazione di una variabile globale ne stabilisce il nome e il tipo ma *non alloca spazio in memoria* per i suoi valori. Una tale dichiarazione globale è premessa dalla parola chiave `extern`. (Cfr. la dichiarazione di una funzione.)
- La definizione di una variabile globale, invece, omette `extern` e comporta l'allocazione della necessaria quantità di memoria. (Cfr. la definizione di una funzione.)
- Tra tutti i file sorgente che compongono il programma vi deve essere *una sola* definizione di ciascuna variabile globale.

Variabili esterne (o globali)

- La dichiarazione di una variabile globale ne stabilisce il nome e il tipo ma *non alloca spazio in memoria* per i suoi valori. Una tale dichiarazione globale è premessa dalla parola chiave `extern`. (Cfr. la dichiarazione di una funzione.)
- La definizione di una variabile globale, invece, omette `extern` e comporta l'allocazione della necessaria quantità di memoria. (Cfr. la definizione di una funzione.)
- Tra tutti i file sorgente che compongono il programma vi deve essere *una sola* definizione di ciascuna variabile globale.
- Vi possono essere invece più dichiarazioni `extern`, se la variabile è usata in più di un file sorgente del programma.

Variabili esterne (o globali)

- La dichiarazione di una variabile globale ne stabilisce il nome e il tipo ma *non alloca spazio in memoria* per i suoi valori. Una tale dichiarazione globale è premessa dalla parola chiave `extern`. (Cfr. la dichiarazione di una funzione.)
- La definizione di una variabile globale, invece, omette `extern` e comporta l'allocazione della necessaria quantità di memoria. (Cfr. la definizione di una funzione.)
- Tra tutti i file sorgente che compongono il programma vi deve essere *una sola* definizione di ciascuna variabile globale.
- Vi possono essere invece più dichiarazioni `extern`, se la variabile è usata in più di un file sorgente del programma.
- Per quanto detto, `extern` si usa soprattutto assieme alla compilazione separata, ossia quando il codice di un programma risiede in più file: non tratteremo l'argomento oltre questi cenni.

Funzioni

- Il campo di visibilità di una funzione è soggetto alle medesime regole che si applicano alle variabili globali.

Funzioni

- Il campo di visibilità di una funzione è soggetto alle medesime regole che si applicano alle variabili globali.
- Ciò perché una funzione non è mai locale a un altro blocco: *il linguaggio C non permette di definire funzioni all'interno di altre funzioni.*

Funzioni

- Il campo di visibilità di una funzione è soggetto alle medesime regole che si applicano alle variabili globali.
- Ciò perché una funzione non è mai locale a un altro blocco: *il linguaggio C non permette di definire funzioni all'interno di altre funzioni.*
- All'interno di un dato file sorgente una funzione è dunque visibile dal punto in cui è dichiarata fino alla fine del file.

Funzioni

- Il campo di visibilità di una funzione è soggetto alle medesime regole che si applicano alle variabili globali.
- Ciò perché una funzione non è mai locale a un altro blocco: *il linguaggio C non permette di definire funzioni all'interno di altre funzioni.*
- All'interno di un dato file sorgente una funzione è dunque visibile dal punto in cui è dichiarata fino alla fine del file.
- Come abbiamo già visto, anche per le funzioni, come per le variabili, si distingue fra definizione e dichiarazione: e la terminologia è coerente.

Funzioni

La parola chiave `extern` si può usare con le funzioni allo stesso modo in cui la si usa con le variabili globali, ma i moderni compilatori C considerano in automatico tutte le funzioni `extern`, e dunque visibili anche al di fuori del file sorgente in cui sono dichiarate. Ne segue che l'uso di `extern` con le funzioni non è solitamente necessario neppure in presenza di compilazione separata, a differenza che per le variabili globali.

Funzioni

La parola chiave `extern` si può usare con le funzioni allo stesso modo in cui la si usa con le variabili globali, ma i moderni compilatori C considerano in automatico tutte le funzioni `extern`, e dunque visibili anche al di fuori del file sorgente in cui sono dichiarate. Ne segue che l'uso di `extern` con le funzioni non è solitamente necessario neppure in presenza di compilazione separata, a differenza che per le variabili globali.

Nota importante

Il saper individuare con certezza i momenti dell'esecuzione del programma in cui il sistema alloca la memoria destinata a contenere i valori di una variabile, e quelli in cui il sistema la dealloca (ossia la toglie dalla disponibilità del programma, perché non più necessaria), è da considerarsi una *competenza imprescindibile del programmatore*. Questo è uno dei motivi per cui è importante impadronirsi delle regole di visibilità che abbiamo appena discusso.

globale.c

```c
#include <stdio.h>
int x=0; //definizione di variabile globale

int main(void)
{
    void procA();

    printf("Main 1: x=%d\n",x); //x globale
    procA();
    printf("Main 2: x=%d\n",x); //x globale
    return 0;
}

void procA()
{
    x++; //x globale
}
```

locale.c

```c
#include <stdio.h>
int x=0; //definizione di variabile globale

int main(void)
{
    void procA();

    printf("Main 1: x=%d\n",x); //x globale
    int x=1;
    printf("Main 2: x=%d\n",x); //x locale
    procA();
    printf("Main 3: x=%d\n",x); //x locale
    return 0;
}
void procA()
{
    printf("A 2: x=%d\n",x); //x globale
    int x=2;
    printf("A 2: x=%d\n",x); //x locale
}
```