Programmazione 1

Lezione 5

Vincenzo Marra

vincenzo.marra@unimi.it

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

29 marzo 2017

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.

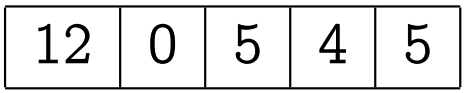
Gli array: introduzione

- 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:

12	0	5	4	5

- Le celle sono numerate consecutivamente a partire da zero:

Indice	0	1	2	3	4
Array	12	0	5	4	5

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

12	0	5	4	5

- Le celle sono numerate consecutivamente a partire da zero:

Indice	0	1	2	3	4
Array	12	0	5	4	5

- Se il nome di questo array è `pippo`, allora per accedere all'elemento i-esimo su 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 12.

- 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 ("allocata", come si dice) al programma, cosa che è comunque 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 indefinito.

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[10000000]);

        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[9-i]=v[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 positivi, 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 "Prog1/IO.h" //Per leggi_int
int main(void)
{
        int i,n;
        //lettura 1
        n=leggi_int("Quanti interi vorrai inserire? ");
        int v[n];
        //lettura 2
        for(i=0; i<n; i++)
                while( (v[i]=leggi_int("Inserisci prossimo intero: "))
                        printf("Solo valori positivi: reinserire.\n");
        //scrittura
        for(i=0; i<n; i++)
        {
                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.

- 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.
- 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.
- 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.
- 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.

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 un uscita dovrebbe essere

```
oaiC
```

rev.c

```c
#include "Prog1/IO.h"
int main(void)
{
        char v[256]; // Conterra' la stringa letta
        if ( (leggi_str("Scrivi qualcosa: ",v)) ==0)
        {
                printf("Errore I/O\n");
                return -1;
        } // v adesso contiene la stringa letta 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
        for (int i=l-1;i>=0;i--)
                printf("%c",v[i]);
        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: Uso della libreria didattica `Prog1` per leggere caratteri, stringhe, interi, reali.

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.
2. Output: Uso di `printf`. Input: Uso della libreria didattica `Prog1` per leggere caratteri, stringhe, interi, reali.

Cosa abbiamo visto fino ad ora?

1. Tipi primitivi (`char`, `int`, `float`, `double`), variabili e costanti, espressioni, assegnamenti.
2. Output: Uso di `printf`. Input: Uso della libreria didattica `Prog1` per leggere caratteri, stringhe, interi, reali.
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: Uso della libreria didattica `Prog1` per leggere caratteri, stringhe, interi, reali.
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: Uso della libreria didattica `Prog1` per leggere caratteri, stringhe, interi, reali.
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: Uso della libreria didattica `Prog1` per leggere caratteri, stringhe, interi, reali.
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'ingresso*).

Le funzioni nella programmazione, in generale

- Astrattamente un programma computa una funzione che associa dati in uscita (*l'output*) a dati in ingresso (*l'ingresso*).
- 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'ingresso*).
- 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 $f : \mathbb{N} \to \mathbb{N}$ definita da $n \in \mathbb{N} \mapsto n^2 + 3$, 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  }
```

- Esempio. Data $f : \mathbb{N} \to \mathbb{N}$ definita da $n \in \mathbb{N} \mapsto n^2 + 3$, 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:

struttura2.c

```
1  int f(int n)
2  {
3          int x = q(n);
4          int y = t(x);
5          return y;
6  }
7
8  int q(int n)
9  {
10         return n*n;
11 }
12 int t(int n)
13 {
14         return n+3;
15 }
```

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, se presa 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, se presa in sé stessa.
- Una volta scritto il codice di una funzione, non è più necessario preoccuparsi dei dettagli della sua implementazione: basterà sapere cosa la funzione fa per poterla usare, ignorando i dettagli di come lo faccia.

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, se presa in sé stessa.
- Una volta scritto il codice di una funzione, non è più necessario preoccuparsi dei dettagli della sua implementazione: basterà sapere cosa la funzione fa per poterla usare, ignorando i dettagli di come lo faccia.
- Per programmare bene in C è essenziale usare opportunamente 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 ne costituisce la 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 ne costituisce la 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 ne costituisce la 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 ne costituisce la 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 ne costituisce la implementazione.

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

funz1.c

```
1  #include <stdio.h>
2
3  void stampa(void) {
4          printf("Ciao Mondo.\n");
5          return;
6  }
7
8  int main(void) {
9          stampa();
10         return 0;
11 }
```

funz2.c

```c
#include <stdio.h>

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

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

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;
}
```

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;
}
```

par1.c

```c
#include <stdio.h>

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

        stampa(-2);
        return 0;
}

void stampa(int n) {
        for (int i=0; i<n; i++)
                printf("Ciao Mondo (%d).\n",i+1);
        return;
}
```

par2.c

```c
#include <stdio.h>

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

        if ( stampa(-2) < 0 )
                printf("Errore.\n");
        return 0;
}

int stampa(int n) {
        if (n<0)
                return -1; //errore
        for (int i=0; i<n; i++)
                printf("Ciao Mondo (%d).\n",i+1);
        return 0; //ok
}
```

varglob.c

```c
#include <stdio.h>

int n; /* var. globale */

int main(void) {
        void funz(void);

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

void funz(void) {
        n++;
        /* posso omettere return se void */
}
```

Il campo di visibilità

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

Il campo di visibilità

Il campo di visibilità — in inglese *scope* — di un nome (di variabile o di funzione) è la parte di programma in cui quel nome può essere usato, in quanto è lì riconosciuto come definito dal compilatore.
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, sono tra loro non correlate.

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, sono tra loro non correlate.
- 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 uno dentro l'altro, occorre stabilire una regola per rendere non ambigui i riferimenti al loro nome comune.

Variabili automatiche (o locali)

Nel caso però in cui due variabili automatiche dallo stesso nome siano dichiarate in due blocchi uno dentro l'altro, occorre stabilire una regola per rendere non ambigui i riferimenti al loro nome comune.

La regola è che la variabile dichiarata più in profondità fa ombra a 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 uno dentro l'altro, occorre stabilire una regola per rendere non ambigui i riferimenti al loro nome comune.

La regola è che la variabile dichiarata più in profondità fa ombra a 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: x=%d\n",x);

        for (int i=0;i<2;i++) //i locale al for
        {
                int x=0; //x locale al corpo del for
                x--;
                printf("For %d: x=%d\n",i,x);
        }

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

        return 0;
}
```

Variabili esterne (o globali)

- Il campo di visibilità di una variabile esterna — 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 esterna — 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 esterna *prima* della sua definizione, oppure in un altro file sorgente, occorre usare la parola chiave `extern`.

Variabili esterne (o globali)

- Il campo di visibilità di una variabile esterna — 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 esterna *prima* della sua definizione, 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 esterna ne stabilisce il nome e il tipo, ma *non alloca spazio in memoria* per i suoi valori.

Variabili esterne (o globali)

- La dichiarazione di una variabile esterna ne stabilisce il nome e il tipo, ma *non alloca spazio in memoria* per i suoi valori.
- La definizione di una variabile esterna, invece, provoca l'allocazione della necessaria quantità di memoria.

Variabili esterne (o globali)

- La dichiarazione di una variabile esterna ne stabilisce il nome e il tipo, ma *non alloca spazio in memoria* per i suoi valori.
- La definizione di una variabile esterna, invece, provoca l'allocazione della necessaria quantità di memoria.
- Tra tutti i file che compongono il programma vi deve essere una sola definizione di ciascuna variabile esterna.

Variabili esterne (o globali)

- La dichiarazione di una variabile esterna ne stabilisce il nome e il tipo, ma *non alloca spazio in memoria* per i suoi valori.
- La definizione di una variabile esterna, invece, provoca l'allocazione della necessaria quantità di memoria.
- Tra tutti i file che compongono il programma vi deve essere una sola definizione di ciascuna variabile esterna.
- 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 esterna ne stabilisce il nome e il tipo, ma *non alloca spazio in memoria* per i suoi valori.
- La definizione di una variabile esterna, invece, provoca l'allocazione della necessaria quantità di memoria.
- Tra tutti i file che compongono il programma vi deve essere una sola definizione di ciascuna variabile esterna.
- Vi possono essere invece più dichiarazioni `extern`, se la variabile è usata in più di un file sorgente del programma.
- L'uso principale delle parola chiave `extern` è dunque in connessione alla compilazione separata — la suddivisione del codice di un programma in più file — un argomento che noi non tratteremo oltre questi cenni.

Funzioni

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

Funzioni

- Il campo di visibilità di una funzione è soggetto alle medesime regole che si applicano alle variabili esterne.
- 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 esterne.
- 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 esterne.
- 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 visto la volta scorsa, anche per le funzioni si distingue fra definizione e dichiarazione: e la terminologia è coerente. Anche in questo senso le funzioni e le variabili esterne sono analoghe.

Funzioni

La parola chiave `extern` si può usare con le funzioni allo stesso modo in cui la si usa con le variabili esterne. Tuttavia, nelle implementazioni moderne dei compilatori C, le funzioni sono già implicitamente considerate `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, a differenza che per le variabili esterne.

globale.c

```c
#include <stdio.h>
int x=0;

int main(void)
{
        void procA();

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

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

locale.c

```c
#include <stdio.h>
int x=0;

int main(void)
{
        void procA();

        printf("Main 1: x=%d\n",x); // x esterna
        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 esterna
        int x=2;
        printf("A 2: x=%d\n",x); // x locale
}
```