Variabili locali

- Il blocco che costituisce il corpo di una funzione/procedura può contenere dichiarazioni di variabili.

  Esempio:

```
void leggiVettore(int v[], int dim)
{
  int i;              /* i E' UNA VARIABILE LOCALE */
  for (i = 0; i < dim; i++) { ... }
}
```

  - sono variabili proprie della funzione
  - hanno tempo di vita limitato alla durata della chiamata
  - più in generale: un identificatore dichiarato nel corpo di una funzione è detto locale alla funzione e non è visibile all'esterno della funzione (ad esempio nel `main`), ma solo nel corpo della stessa

- In realtà, ciò non è altro che un caso particolare di regole generali che governano la visibilità e il tempo di vita degli identificatori di un programma.

Struttura generale di un programma C

- parte direttiva
- parte dichiarativa globale che comprende:
  - dichiarazioni di costanti
  - dichiarazioni di tipi (li vedremo . . .)
  - dichiarazioni di variabili (variabili globali)
  - prototipi di funzioni/procedure
- il programma principale (`main`)
- le definizioni di funzioni/procedure

Esempio

```
#include <stdio.h>       /* parte direttiva */
#define LUNG 10

int i = 1;               /* variabili globali */
int j = 2;

int Q(int);              /* prototipi di funzioni e procedure */
void P(int *);

main()                  /* programma principale */
{
int x = 10;
char c = 'a';
x = Q(x);
P(&x);
}

int Q(int v) { ... }    /* definizioni di funzioni e procedure */
void P(int *z) { ... }
```


Blocchi

- il corpo di una funzione/procedura, cosi' come il corpo del programma principale, è un blocco.
- In C un blocco è costituito da
  - una parte dichiarativa (può non esserci)
  - una parte esecutiva (sequenza di istruzioni)
- Nel `main` o nel corpo delle funzioni possono comparire diversi blocchi, che possono essere
  - annidati: un blocco è una delle istruzioni di un altro blocco
  - paralleli: blocchi che fanno parte della medesima sequenza di istruzioni

```
{
   int x;
   x = 10;
   {
     int z;
     z = 20 ;
     ...
   }
   ...
}
```

```
{
   int x;
   x = 10;
   ...
}
{
   int z;
   z = 20;
   ...
}
```

- Anche la parte esecutiva del programma principale e di una funzione/procedura è un blocco
- Gli identificatori dichiarati nella parte dichiarativa di un blocco sono detti nomi locali del blocco e devono essere tutti diversi tra loro
  - nel caso di una funzione/procedura, fanno parte dei nomi locali anche gli identificatori utilizzati per i parametri formali

Esempio:

```
{
int x;   /* NO! identificatore x dichiarato   */
char x;  /* due volte nello stesso blocco     */
...
}

void p(int x, char y)
{
int x;      /* NO! identificatore x gia' usato per un parametro formale */
...
}
```


- In blocchi diversi possono essere utilizzati gli stessi identificatori

Esempio:

```
main()
{
int x;     /* x, y: variabili locali del main */
int y;
...
   {
     char x;    /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;         /*x,y: variabili locali della procedura p  */
...
}
```

- Un programma C può avere una struttura molto complessa a seguito dell'uso di funzioni, procedure e blocchi.
- È necessario definire regole precise per regolamentare l'uso dei nomi utilizzati all'interno di un programma.
- A questo scopo introduciamo alcune definizioni utili.

  Ambiente globale: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa globale del programma

  Ambiente locale di una funzione: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa della funzione e nella sua intestazione

  Ambiente locale di un blocco: è l'insieme di tutti gli elementi (nomi) dichiarati nella parte dichiarativa del blocco
- Quanto detto informalmente in precedenza può essere meglio precisato:

  $\Longrightarrow$ è possibile dichiarare più volte lo stesso identificatore (anche con significati diversi) purché in ambienti diversi
- Se ciò evita il proliferare di identificatori, causa il problema di stabilire il significato di un riferimento ad un identificatore in un generico punto del programma

Esempio: Riprendiamo l'esempio precedente

```
main()
{
int x;     /* x, y: variabili locali del main */
int y;
...
   {
     char x;    /* x: variabile locale del blocco annidato */
     ...
   }
...
}

void p(int x)
{
int y;         /*x,y: variabili locali della procedura p  */
...
}
```

- Se in un punto del programma viene eseguita l'istruzione `x = ...`, a quale delle tre dichiarazioni di `x` ci si riferisce?
- Dipende dal punto in cui si trova tale assegnamento e dalle regole di visibilità (o regole di scoping).

Regole di visibilità

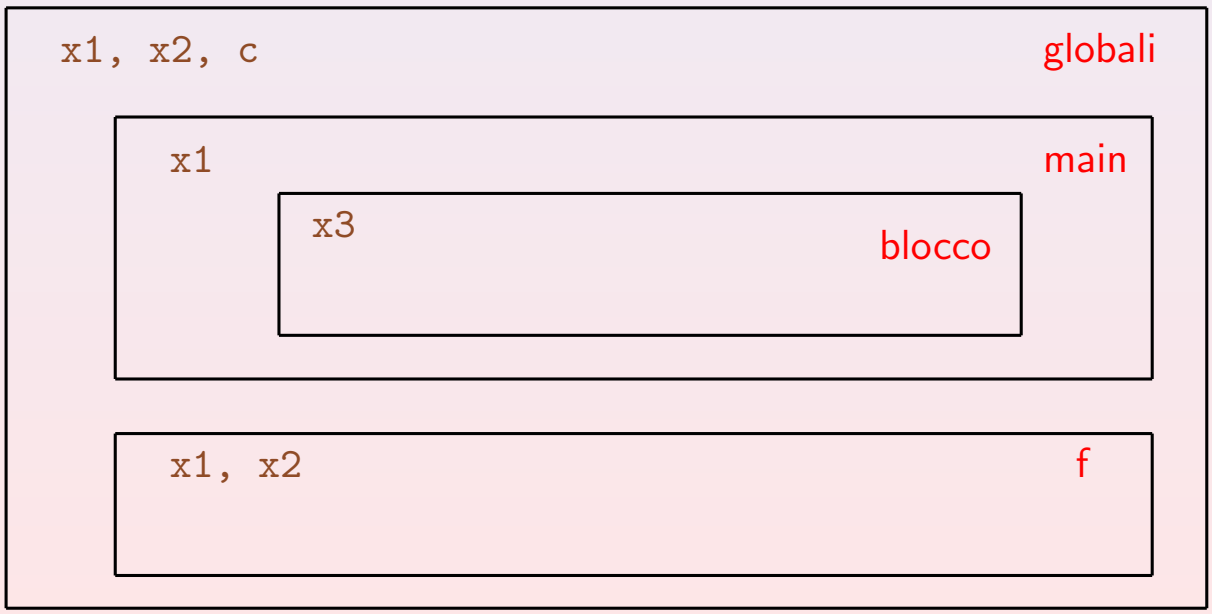
- Ogni identificatore deve essere dichiarato prima di essere utilizzato.
- Gli identificatori dichiarati nell'ambiente globale sono visibili in tutte le funzioni e in tutti i blocchi del programma.
  N.B. Gli identificatori predefiniti del linguaggio si intendono parte dell'ambiente globale.
- Gli identificatori dichiarati nell'ambiente locale di una funzione sono visibili dalla dichiarazione fino alla fine del corpo della funzione (compresi eventuali blocchi in esso contenuti).
- Gli identificatori dichiarati nell'ambiente locale di un blocco sono visibili dalla dichiarazione fino alla fine del blocco (compresi eventuali blocchi in essa contenuti).
- Ogni uso di un identificatore è associato univocamente a una dichiarazione dello stesso.
- Se l'uso di un identificatore è visibile da più dichiarazioni per lo stesso identificatore, la dichiarazione ad esso associata è la più vicina tra di esse.

Esempio:

```
int x1=10, x2=20;
char c='a';

int f(int);

main()
{
int x1=30;    /* nasconde la variabile globale x1 */
x2 = x1+x2;   /* x1 e' quella locale, x2 e' globale */
printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=30  x2=50  */
  { int x3=50;
    x1=f(x3); /* x1 e' quella locale al primo blocco */
    printf("x1=%d   x2=%d\n", x1, x2);   /* stampa x1=150  x2=50 */
  }
}

int f(int x1)  /* nasconde la variabile globale x1 */
{ int x2;       /* nasconde la variabile globale x2 */
  x2 = x1 + 100;  /* x1 e' il parametro formale, x2 la var. locale */
  return x2;
}
```

Rappresentazione Grafica: Modello a contorni

- Si rappresenta ogni ambiente mediante un rettangolo con gli identificatori in esso contenuti.



Durata delle variabili

- Una variabile ha un suo tempo di vita.
  - viene creata (ovvero ad essa viene riservata uno spazio di memoria)
  - viene (o può essere) distrutta (ovvero viene rilasciato il corrispondente spazio di memoria).
- Si distinguono due classi di variabili:
  - variabili automatiche: vengono create ogni volta che si entra nel loro ambiente di visibilità e vengono distrutte all'uscita di tale ambiente
    - es. variabili locali di un blocco: vengono create all'ingresso del blocco { distrutte all'uscita dal blocco }
    - es. variabili locali di una funzione: vengono create al momento della chiamata e distrutte all'uscita
  - variabili statiche: vengono create una sola volta e vengono distrutte solo al termine dell'esecuzione del programma (non ne faremo uso ...)
- N.B. nel caso di funzioni/blocchi eseguiti più volte (es. funzione chiamata in punti diversi, blocco all'interno di un ciclo):
  le variabili automatiche corrispondenti possono essere associate di volta in volta a locazioni di memoria diverse, quindi
  il loro valore non persiste tra una esecuzione e la successiva

Gestione della memoria a tempo di esecuzione (run-time)

- Il codice macchina e i dati risiedono entrambi in memoria, ma in zone separate:
  - la memoria per il codice macchina è fissata a tempo di compilazione
  - la memoria per i dati (in particolare per le variabili automatiche) cresce e decresce dinamicamente durante l'esecuzione: viene gestita a pila
- Una pila (o stack) è una struttura dati con accesso LIFO: Last In First Out = l'ultimo entrato è il primo ad uscire (es.: pila di piatti da lavare).
- Il sistema gestisce in memoria la pila dei record di attivazione (RDA)
  - per ogni chiamata di funzione viene creato un nuovo RDA in cima alla pila
  - al termine della chiamata della funzione il RDA viene rimosso dalla pila
- Ogni RDA contiene:
  - le locazioni di memoria per i parametri formali (se presenti)
  - le locazioni di memoria per le variabili locali (se presenti)
  - altre informazioni che non analizziamo
- Anche gli ambienti locali dei blocchi vengono allocati/deallocati sulla pila.

Esempio:

```
int f(int);
main()
{
  int x, y, z;
  x=10;
  y=20;          /* blocco principale */                    ◂ PUNTO 1
  z = f(x);      /* prima chiamata di f */                  ◂ PUNTO 2
    {
    int x=50;    /* uscita da f e ingresso nel blocco annidato*/  ◂ PUNTO 3
    y=f(x);      /* seconda chiamata di f */                ◂ PUNTO 4
    z=y;         /* uscita da f */                          ◂ PUNTO 5
    }
  ...            /* uscita dal blocco */                    ◂ PUNTO 6
}
int f(int a)
{
  int z;
  z = a + 1;
  return z;
}
```

Evoluzione della pila

x	10
y	20
z	?

PUNTO 1

Evoluzione della pila

a	10
z	?

x	10
y	20
z	?

PUNTO 2

Evoluzione della pila

x	50

x	10
y	20
z	11

PUNTO 3

Evoluzione della pila

a	50
z	?

x	50

x	10
y	20
z	11

PUNTO 4

Evoluzione della pila

x	50

x	10
y	51
z	11

PUNTO 5

Evoluzione della pila

x	10
y	51
z	51

PUNTO 6

Variabili statiche: un esempio d'uso

- Una variabile statica, una volta creata, rimane in vita per tutto il tempo di esecuzione del programma.
  Esempio: `f(void) { static int x; ... }`
  - la variabile viene inizializzata alla prima attivazione della funzione
  - conserva il suo valore tra attivazioni successive
  - è locale, quindi visibile solo all'interno della funzione in cui è dichiarata

  Esempio: Funzione che ritorna il numero di volte che è stata attivata.

```
int fun1(void) {
   static int conta = 0;
       /* variabile locale statica visibile solo in fun1;
          contatore del numero di attivazioni di fun1    */
   .....
   conta++;
   return conta;
   }
```


Funzioni per la manipolazione di stringhe

Una funzione per uguaglianza di stringhe

```
boolean uguali(char * str1, char * str2)
/* Restituisce 'true' se le due stringhe str1 e str2
   sono uguali, 'false' altrimenti. */
{
  int i = 0;
  while (str1[i] == str2[i] && str1[i] != '\0'
         && str2[i] != '\0')
    i++;
  if (str1[i] == '\0' && str2[i] == '\0')
    return true;
  else return false;
}
```

Esercizio

Lunghezza di una stringa.

```
int lunghezza1(char * stringa)
  /* Restituisce la lunghezza della stringa
     passata come parametro.  */
{
  int i = 0;
  while (stringa[i] != '\0')
    i++;
  return i;
}
```

In realtà queste funzioni, come molte altre, sono disponibili nella libreria `<string.h>`

Per utilizzarle è *necessario:* `#include <string.h>`

Funzioni di libreria in <string.h>

Lunghezza di una stringa

```
unsigned strlen(const char *str);
```

Esempio: `strlen("abc")` restituisce il valore `3` mentre `strlen("")` restituisce il valore `0`.

Funzioni di confronto

```
int strcmp(const char *s1, const char *s2);
```

- confronta le stringhe `s1` ed `s2`
- restituisce:
  - 0 se `s1` = `s2`
  - un valore < 0 se `s1` $<$ `s2`
  - un valore > 0 se `s1` $>$ `s2`
- il confronto è quello lessicografico: i caratteri vengono confrontati uno ad uno, ed il primo carattere diverso (o la fine di una delle due stringhe) determina il risultato
- per il confronto tra due caratteri viene usato il codice numerico corrispondente

Esempio di confronto tra stringhe

```
char *s1 = "abc";
char *s2 = "abx";
char *s3 = "abc altro";
printf("%d\n", strcmp(s1, s1));
printf("%d\n", strcmp(s1, s2));
printf("%d\n", strcmp(s1, s3));
printf("%d\n", strcmp(s2, s1));
```

Stampa:

```
0
-1
-1
1
```


Attenzione: per verificare l'uguaglianza di due stringhe usando `strcmp()` bisogna confrontare il risultato con 0

Esempio:

```
if (strcmp(s1, s2) == 0)
  printf("uguali\n");
else
  printf("diverse\n");
```

La funzione `strncmp()`

`int strncmp(const char *s1, const char *s2, size_t n);`

confronta al più `n` caratteri di `s1` ed `s2`

Esempio:

```
char *s1 = "abc";
char *s3 = "abc altro";

printf("%d\n", strncmp(s1, s3, 3));
```

Stampa `0`, poiché i primi tre caratteri delle due stringhe sono uguali.

Funzioni di copia

```
char *strcpy(char *dest, const char *src);
```

copia la stringa `src` nel vettore `dest`
restituisce il valore puntatore `dest`
`dest` dovrebbe essere sufficientemente grande da contenere `src`
se `src` e `dest` si sovrappongono, il comportamento di `strcpy` è indefinito

Esempio:

```
char a[10], b[10];
char *x = "Ciao";
char *y = "mondo";
strcpy(a, x);    strcpy(b, y);
printf("%s\n", a);          printf("%s\n", b);
```

Stampa:

```
Ciao
mondo
```



```
char *strncpy(char *dest, const char *src, size_t n);
```

copia un massimo di `n` caratteri della stringa `src` nel vettore `dest`
(`size_t` è il tipo del valore restituito da **sizeof**, ovvero **unsigned long** o **unsigned**, dipendente dal sistema)
restituisce il valore puntatore di dest
Attenzione: il carattere `'\0'` finale di `src` viene copiato *solo* se `n` è $\geq$ della lunghezza di `src+1`

Esempio:

```
char a[10], b[10];
char *x = "Ciao";
char *y = "mondo";

strncpy(a, x, 5);
printf("%s\n", a);
strncpy(b, y, 5);
b[5] = '\0';
printf("%s\n", b);
```

Funzioni di concatenazione

```
char *strcat(char *dest, const char *src);
```

accoda la stringa `src` a quella nel vettore `dest` (il primo carattere di `src` si sostituisce al `'\0'` di `dest`) e termina `dest` con `'\0'`
restituisce il valore di `dest`
`dest` dovrebbe essere sufficientemente grande da contenere tutti i caratteri di `dest`, di `src`, ed il `'\0'` finale
se `src` e `dest` si sovrappongono, il comportamento di `strcat` è indefinito

Esempio:

```
char a[10], b[10];
char *x = "Ciao",  *y = "mondo";
strcpy(a, x);
strcat(a, y);
printf("%s\n", a);
```


stampa: `Ciaomondo`

Conversione di stringhe

Convertono le strighe formate da cifre in valori interi ed in virgola mobile.

Funzioni `atoi`, `atol`, `atof`

```
int atoi(const char *str);
```

converte la stringa puntata da `str` in un `int`
la stringa deve contenere un numero intero valido
in caso contrario il risultato è indefinito
spazi bianchi iniziali vengono ignorati
il numero può essere concluso da un qualsiasi carattere che non è valido in un numero intero (spazio, lettera, virgola, punto, ...)

Esempio: file `stringhe/strtonum.c`

`atoi("123")`	restituisce l'intero 123
`atoi("123.45")`	restituisce l'intero 123 (`".45"` viene ignorato)
`atoi("~~123.45")`	restituisce l'intero 123 (`"~~"` e `".45"` ignorati)

```
long atol(const char *str);
```
analoga ad `atoi`, solo che restituisce un `long int`

```
double atof(const char *str);
```
analoga ad `atoi` ed `atol`, solo che restituisce un `double`
il numero può essere concluso da un qualsiasi carattere non valido in un numero in virgola mobile

Esempio: file `stringhe/strtonum.c`

`atof("123.45")` restituisce 123.45
`atof("1.23xx")` restituisce 1.23
`atof("1.23.45")` restituisce 1.23

Funzioni di libreria per I/O di stringhe e caratteri in <stdio.h>

Input/output di caratteri

```
int getchar(void);
```
legge il prossimo carattere da standard input e lo restituisce

```
int putchar(int c);
```
manda `c` in standard output
restituisce `EOF` (-1) se si verifica un errore

Input/output di stringhe

`char *gets(char *str);`

legge i caratteri da standard input e li inserisce nel vettore `str`
la lettura termina con un `'\n'` o un `EOF`, che *non* viene inserito nel vettore
la stringa nel vettore viene terminata con `'\0'`
Attenzione: non c'è alcun modo di limitare la lunghezza della sequenza immessa $\Longrightarrow$ vettore allocato potrebbe non bastare

`int puts(const char *str);`

manda la stringa `str` in standard output (inserisce un `'\n'` alla fine)

Vettori di stringhe

- Un vettore di stringhe è un puntatore a puntatore a `char`
- Dichiarazioni di vettori di stringhe con inizializzazione:

  ```
  char *colori[4] = {"rosso", "giallo", "verde", "blu"};
  ```

  $\Longrightarrow$ vettore di quattro puntatori a quattro stringhe costanti (di lunghezza 6, 7, 6, 4).
- È equivalente ad inizializzare i quattro puntatori separatamente:

  ```
  char *colori[4];
  colori[0] = "rosso";      colori[1] = "giallo";
  colori[2] = "verde";      colori[3] = "blu";
  ```

- Dichiarazioni alternative (non sempre equivalenti):

  ```
  char colori[4][15] = {"rosso", "giallo", "verde", "blu"};
  char colori[4][] = {"rosso", "giallo", "verde", "blu"};
  char **colori = {"rosso", "giallo", "verde", "blu"};
  ```

Programmi con argomenti passati da linea di comando

- Quando eseguiamo un programma da uno shell, possiamo volergli passare degli argomenti dalla riga di comando (si pensi ai comandi di Linux). Per esempio, se abbiamo scritto un programma `copy`, per copiare `file1` in `file2` vorremmo invocarlo come

  ```
  copy file1 file2
  ```

- Per poter usare gli argomenti con cui il programma è stato lanciato, dobbiamo usare due parametri nel `main`

  ```
  int main(int argc, char *argv[]) { ... }
  ```

  - `argc` ... numero di argomenti con cui il programma è stato lanciato più 1
  - `argv` ... vettore di `argc` stringhe che compongono la riga di comando
    - `argv[0]` è il nome del programma stesso
    - `argv[1]`, ..., `argv[argc-1]` sono gli argomenti (separati da spazio)
  - ogni `argv[i]` è una stringa (terminata da `'\0'`)
  - `argc` e `argv` vengono inizializzati dal sistema operativo (le stringhe di `argv` sono allocate dinamicamente)

Esempio: Stampa del numero di argomenti ricevuti da linea di comando.

```
#include <stdio.h>
 main(int argc, char *argv[])
{
  printf("Ho ricevuto %d argomenti\n", argc-1);
}
```

Compilandolo nel file `argnum` e eseguendolo:

```
> argnum primo secondo terzo
> Ho ricevuto 3 argomenti
```

Stampa degli argomenti ricevuti da linea di comando

```
#include <stdio.h>
main(int argc, char* argv[])
{
  int i;
  printf("Ho ricevuto %d argomenti\n", argc-1);
  printf("Questi argomenti sono:\n");
  for (i = 0; i <= argc-1; i++)
    printf("argv[%d]=%s\n", i,argv[i]);
}
```

Compilando nel file `myecho`, ed eseguendolo si ottiene:

```
>myecho is for apple
Ho ricevuto 3 argomenti
Questi argomenti sono:
argv[0]=myecho
argv[1]=is
argv[2]=for
argv[3]=apple
```


Esercizi

Esercizio
Fornire un'implementazione delle funzioni di libreria su caratteri e stringhe.

Esercizio
Programma che legge una data nel formato "12/05/2002" e la converte nel formato "12 maggio 2002".

Esercizio
Programma che legge un numero tra 0 e 999 999 e lo stampa in lettere.