Programmazione in C

Unità
Funzioni

Funzioni

- Tipi di dato
- Funzioni in C
- Modifica dei parametri
- Parametri "by reference"
- La funzione `main()`
- Esercizi proposti
- Sommario

Riferimenti al materiale

- Testi
  - Kernighan & Ritchie: capitoli 2 e 4
  - Cabodi, Quer, Sonza Reorda: capitoli 3 e 7
  - Dietel & Dietel: capitolo 5
- Dispense
  - Scheda: "Tipi di dato in C"
  - Scheda: "Funzioni in C"

Funzioni

Tipi di dato

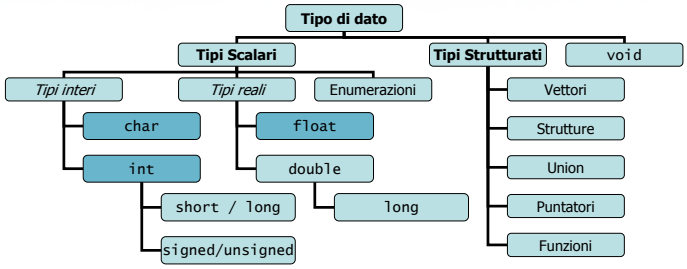
Tipi di dato

- I tipi scalari in C
- Input/output dei tipi scalari
- Conversioni di tipo

Tipi di dato

I tipi scalari in C

Il sistema dei tipi C



I tipi interi in C

Tipo	Descrizione	Esempi
`char`	Caratteri ASCII	`'a' '7' '!'`
`int`	Interi…	`+2 -18 0 +24221`
`short int`	… con meno bit	
`long int`	… con più bit	
`unsigned int`	Interi senza segno…	`0 1 423 23234`
`unsigned short int`	… con meno bit	
`unsigned long int`	… con più bit	

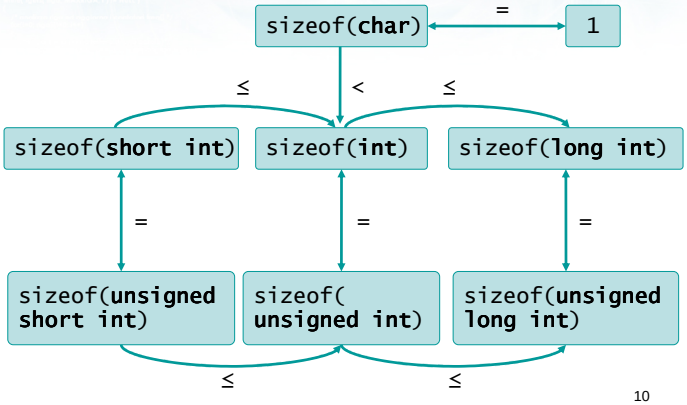
Quanti bit?

- Lo standard C non specifica l'ampiezza, in bit, dei tipi di dato fondamentali
- L'operatore `sizeof` può essere usato per ricavare una costante pari al numero di byte occupato da ciascun tipo

```
sizeof(char)
```

```
sizeof(int)
```

Specifiche del C



Intervallo di rappresentazione

Tipo	Min	Max
char	CHAR_MIN	CHAR_MAX
int	INT_MIN	INT_MAX
short int	SHRT_MIN	SHRT_MAX
long int	LONG_MIN	LONG_MAX
unsigned int	0	UINT_MAX
unsigned short int	0	USHRT_MAX
unsigned long int	0	ULONG_MAX

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


Compilatori a 32 bit

Tipo	N. Bit	Min	Max
char	8	-128	127
int	32	-2147483648	2147483647
short int	16	-32768	32767
long int	32	-2147483648	2147483647
unsigned int	32	0	4294967295
unsigned short int	16	0	65536
unsigned long int	32	0	4294967295

I tipi reali in C

Tipo	Descrizione
float	Numeri reali in singola precisione
double	Numeri reali in doppia precisione
long double	Numeri reali in massima precisione

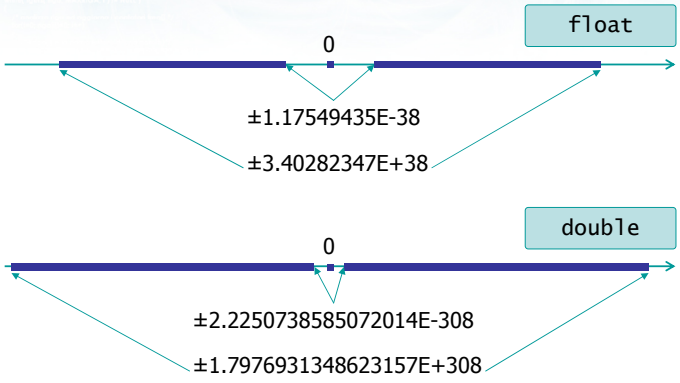
$$A = \overbrace{\pm}^{segno}\ \underbrace{1.mmmmm}_{mantissa} \times 2^{\overbrace{\pm eeeeee}^{esponente}}$$

Numero di bit

Tipo	Dimensione	Mantissa	Esponente
float	32 bit	23 bit	8 bit
double	64 bit	53 bit	10 bit
long double	≥ double		

$$A = \overbrace{\pm}^{segno}\ \underbrace{1.mmmmm}_{mantissa} \times 2^{\overbrace{\pm eeeeee}^{esponente}}$$

Intervallo di rappresentazione



Tipi di dato

Input/output dei tipi scalari

Input/output

- I diversi tipi scalari visti sono utilizzabili con le normali funzioni `scanf`/`printf`, adottando degli specifici indicatori di formato
- Utilizzando la funzione `gets` per l'input, si possono usare le funzioni di conversione `ato...`

Specificatori di formato

Tipo	scanf	printf
char	%c %[...]	%c %d
int	%d	%d
short int	%hd	%hd %d
long int	%ld	%ld
unsigned int	%u %o %x	%u %o %x
unsigned short int	%hu	%hu
unsigned long int	%lu	%lu
float	%f	%f %g
double	%lf	%f %g

Funzioni di conversione

```
char line[80] ;
int x ;

gets(line) ;
x = atoi(line) ;
```

```
char line[80] ;
float x ;

gets(line) ;
x = atof(line) ;
```

```
char line[80] ;
long int x ;

gets(line) ;
x = atol(line) ;
```

```
char line[80] ;
double x ;

gets(line) ;
x = atof(line) ;
```


Tipi di dato

Conversioni di tipo

Conversioni di tipo (1/2)

- Nel linguaggio C è possibile combinare, nella stessa espressione, variabili di tipo diverso
  - I due operandi di un operatore aritmetico possono avere tipi diversi

```
int a ;
long int b, c ;
c = b + a ;
```

```
double prod ;
float v[N] ;
prod = prod * v[i] ;
```

Conversioni di tipo (2/2)

- La variabile di destinazione di un'assegnazione può avere tipo diverso dal tipo dell'espressione

```
int a ;
long int b ;
b = a ;
```

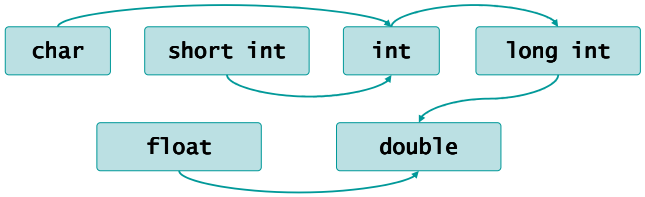
```
double prod ;
float v[N] ;
prod = v[0] ;
```


Tipologie di conversioni

- Per calcolare tali tipi di espressioni, il linguaggio C applica tre tipi di conversioni:
  - Conversioni "automatiche" verso il tipo più capiente, basate sul principio di **promozione del tipo**
  - Arrotondamenti e troncamenti, in caso di assegnazioni "forzate" a tipi meno capienti
  - Conversioni "esplicite", basate sull'operatore di **typecasting**

Promozione del tipo

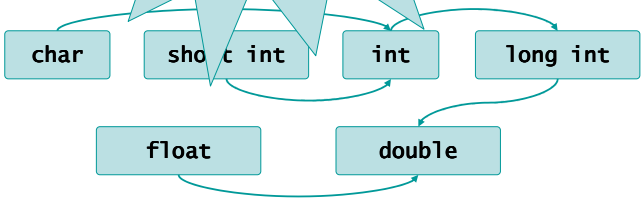
- Se i due operandi di un operatore aritmetico hanno tipo diverso, l'operando del tipo più limitato viene convertito al tipo dell'operando più esteso



Promozione del tipo

- Se i d ore aritmetico
  lim po più
  est dell'operando più

Non si perde mai precisione.
Il C converte automaticamente verso i tipi più capienti



Troncamento del risultato

- Nell'operatore di assegnazione ci possono essere 3 casi:

```
var = expr ;
```

  - La variabile destinazione ha lo stesso tipo dell'espressione calcolata
  - La variabile destinazione ha un tipo più ampio del tipo dell'espressione calcolata
    - Si promuove il tipo dell'espressione al tipo della variabile destinazione
  - La variabile destinazione ha un tipo più ristretto del tipo dell'espressione calcolata
    - Si approssima il tipo dell'espressione al tipo della variabile destinazione, perdendo precisione

Conversioni esplicite

- Qualora si vogliano modificare le regole predefinite di promozione e troncamendo dei tipi, il C mette a disposizione un operatore di **conversione esplicita di tipo**
  - Typecasting

```
(nuovotipo)expr
```

  - Converte l'espressione `expr` dal suo tipo nativo, al tipo desiderato `nuovotipo`
    - Più capiente
    - Meno capiente: troncamento o approssimazione

Esempio 1

```
double media ;
int somma, N ;

media = somma / N ; /* no */

media = (double)somma / N ;
```


Esempio 2

```
int voto ;
float parte1, parte2, parte3 ;


voto = (int) ((parte1 +
        parte2 + parte3)/3) ;
```


Esempio 3

```
int voto ;
float parte1, parte2, parte3 ;
float media ;

/* arrotondamento all'intero
   più vicino */

media = (parte1 +
         parte2 + parte3)/3 ;

voto = (int) (media + 0.5) ;
```

Funzioni

Funzioni in C

Funzioni in C

- Il concetto di funzione
- Parametri formali e attuali
- Il valore di ritorno
- Definizione e chiamata di funzioni
- Passaggio dei parametri
- Corpo della funzione

Funzioni in C

Il concetto di funzione

Strategie di programmazione

- Riuso di codice esistente
  - Funzionalità simili in programmi diversi
  - Funzionalità ripetute all'interno dello stesso programma
  - Minore tempo di sviluppo
  - Frammenti di codice già verificati
  - Utilizzo di parti di codice scritte da altri
    - Funzioni di libreria
    - Sviluppo collaborativo

Come riusare il codice? (1/3)

- Copia-e-incolla
  - Semplice, ma poco efficace
  - Occorre adattare il codice incollato, ritoccando i nomi delle variabili e costanti utilizzati
  - Se si scopre un errore, occorre correggerlo in tutti i punti in cui è stato incollato
  - Nel listato finale non è evidente che si sta riutilizzando la stessa funzionalità
  - Occorre disporre del codice originario
  - Occorre capire il codice originario

Come riusare il codice? (2/3)

- Definizione di funzioni
  - Dichiarazione esplicita che una certa funzionalità viene utilizzata in più punti
  - Si separa la definizione della funzionalità rispetto al punto in cui questa viene utilizzata
  - La stessa funzione può essere usata più volte con parametri diversi

Come riusare il codice? (3/3)

- Ogni miglioramento o correzione è automaticamente disponibile in tutti i punti in cui la funzione viene usata
- Nel listato finale è evidente che si sta riutilizzando la stessa funzionalità
- Non occorre disporre del codice originario
- Non occorre capire il codice originario

Principio di funzionamento (1/3)

```
int main(void)
{
  int x, y ;

/* leggi un numero
   tra 50 e 100 e
   memorizzalo
   in x */
/* leggi un numero
   tra 1 e 10 e
   memorizzalo
   in y */

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


Principio di funzionamento (2/3)

```
int main(void)
{
  int x, y ;

  x = leggi(50, 100) ;
  y = leggi(1, 10) ;

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


Principio di funzionamento (3/3)

```
int main(void)
{
  int x, y ;

  x = leggi(50, 100) ;
  y = leggi(1, 10) ;

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

```
int leggi(int min,
          int max)
{
  int v ;

  do {
    scanf("%d", &v) ;
  } while( v<min ||
           v>max) ;

  return v ;
}
```


Principio di funzionamento (3/3)

```
int main(void)
{
  int x, y ;

  x = leggi(50, 100) ;
  y = leggi(1, 10) ;

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

min=50 max=100

```
int leggi(int min,
          int max)
{
  int v ;

  do {
    scanf("%d", &v) ;
  } while( v<min ||
           v>max) ;

  return v ;
}
```


Principio di funzionamento (3/3)

```
int main(void)
{
  int x, y ;

  x = leggi(50, 100) ;
  y = leggi(1, 10) ;

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

Chiamante

min=50 max=100

```
int leggi(int min,
          int max)
{
  int v ;

  do {
    scanf("%d", &v) ;
  } while( v<min ||
           v>max) ;

  return v ;
}
```

Chiamato

Principio di funzionamento (3/3)

```
int main(void)
{
  int x, y ;

  x = leggi(50, 100) ;
  y = leggi(1, 10) ;

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

min=1 max=10

```
int leggi(int min,
          int max)
{
  int v ;

  do {
    scanf("%d", &v) ;
  } while( v<min ||
           v>max) ;

  return v ;
}
```


Sommario

- La definizione di una **funzione** delimita un frammento di codice riutilizzabile più volte
- La funzione può essere **chiamata** più volte
- Può ricevere dei **parametri** diversi in ogni chiamata
- Può restituire un **valore di ritorno** al **chiamante**
  - Istruzione `return`

Miglioramento della funzione

```
int leggi(int min, int max)
{
  char riga[80] ;
  int v ;

  do {
    gets(riga) ;
    v = atoi(riga) ;
    if(v<min) printf("Piccolo: min %d\n", min) ;
    if(v>max) printf("Grande: max %d\n", max) ;
  } while( v<min || v>max) ;

  return v ;
}
```


Funzioni in C

Parametri formali e attuali

Parametri di una funzione

- Le funzioni possono ricevere dei parametri dal proprio chiamante
- Nella funzione:
  - Parametri **formali**
  - Nomi "interni" dei parametri

```
int leggi(int min,
          int max)
{
   ...
}
```


Parametri di una funzione

- Le funzioni possono ricevere dei parametri dal proprio chiamante
- Nella funzione:
  - Parametri **formali**
  - Nomi "interni" dei parametri
- Nel chiamante:
  - Parametri **attuali**
  - Valori effettivi (costanti, variabili, espressioni)

```
int leggi(int min,
          int max)
{
   ...
}
```

```
int main(void)
{
   ...
   y = leggi(1, 10);
   ...
}
```

Parametri formali (1/2)

- Uno o più parametri
- Tipo del parametro
  - Tipo scalare
  - Vettore o matrice
- Nome del parametro

```
int leggi(int min,
          int max)
{
   ...
}
```

- Nel caso in cui la funzione non abbia bisogno di parametri, si usa la parola chiave `void`

```
int stampa_menu(void)
{
   ...
}
```


Parametri formali (2/2)

- Per parametri vettoriali esistono 3 sintassi alternative
  - `int v[]`
  - `int v[MAX]`
  - `int *v`
- Per parametri matriciali
  - `int m[RIGHE][COL]`
  - `int m[][COL]`

```
int leggi(int v[])
{
   ...
}
```

```
int sudoku(int m[9][9])
{
   ...
}
```


Avvertenza (1/2)

- Il valore della dimensione del vettore (es. `MAX`)
  - Viene totalmente ignorato dal meccanismo di chiamata
  - Non sarebbe comunque disponibile alla funzione chiamata
  - Meglio per chiarezza ometterlo
  - Si suggerisce di passare un ulteriore parametro contenente l'occupazione del vettore

Avvertenza (2/2)

- Nel caso di matrici
  - Il secondo parametro (es. `COL`) è obbligatorio e deve essere una costante
  - Il primo parametro viene ignorato e può essere omesso
  - Per matrici pluri-dimensionali, occorre specificare tutti i parametri tranne il primo

Parametri attuali (1/2)

- Uno o più valori, in esatta corrispondenza con i parametri formali
- Tipi di dato compatibili con i parametri formali
- È possibile usare
  - Costanti
  - Variabili
  - Espressioni

```
int main(void)
{
   y = leggi(1, 10);
}
```

```
int main(void)
{
   y = leggi(a, b);
}
```

```
int main(void)
{
   y = leggi(a+b+1,
             c*2);
}
```

Parametri attuali (2/2)

- I nomi delle variabili usate non hanno alcuna relazione con i nomi dei parametri formali

```
int main(void)
{
   ...
   y = leggi(a, b);
   ...
}
```

min=a max=b

- Le parentesi sono sempre necessarie, anche se non vi sono parametri

```
int main(void)
{
   ...
   y = stampa_menu();
   ...
}
```

Funzioni in C

Il valore di ritorno

Valore di ritorno

- Ogni funzione può ritornare un valore al proprio chiamante
- Il tipo del valore di ritorno deve essere **scalare**
- L'istruzione `return`
  - Termina l'esecuzione della funzione
  - Rende disponibile il valore al chiamante

```
int leggi(int min,
          int max)
{
  int v ;

  scanf("%d", &v) ;

  return v ;
}
```

```
int main(void)
{
  ...
  y = leggi(a, b) ;
  ...
}
```

Tipo del valore di ritorno

- Valore scalare
  - `char`, `int` (o varianti), `float`, `double`
- Tipi avanzati
  - Puntatori, struct
- Nessun valore ritornato
  - `void`

```
double sqrt(double a)
{
   ...
}
```

```
void stampa_err(int x)
{
   ...
}
```

L'istruzione `return` (1/2)

- Restituisce il valore
  - Costante
  - Variabile
  - Espressione
- Il tipo deve essere compatibile con il tipo dichiarato per il valore di ritorno
- Sintassi
  - `return x ;`
  - `return(x) ;`
- L'esecuzione della funzione viene interrotta

L'istruzione `return` (2/2)

- Per funzioni che non ritornano valori (`void`):
  - `return ;`
- Il raggiungimento della fine del corpo della funzione } equivale ad un'istruzione `return` senza parametri
  - Permesso solo per funzioni `void`
- Per funzioni non-`void`, è obbligatorio che la funzione ritorni **sempre** un valore

Nel chiamante...

- Il chiamante può:
  - Ignorare il valore ritornato
    - `scanf("%d", &x) ;`
  - Memorizzarlo in una variabile
    - `y = sin(x) ;`
  - Utilizzarlo in un'espressione
    - `if ( sqrt(x*x+y*y)>z ) ...`

Convenzioni utili

- Le funzioni di tipo matematico ritornano sempre un valore `double`
- Le funzioni che non devono calcolare un valore (ma effettuare delle operazioni, per esempio) ritornano solitamente un valore `int`
  - Valore di ritorno == 0 ⇒ tutto ok
  - Valore di ritorno != 0 ⇒ si è verificato un errore
- Molte eccezioni importanti: `strcmp`, `scanf`, ...

Funzioni in C

Definizione e chiamata di funzioni

Sintassi C per le funzioni

- Il linguaggio C prevede 3 distinti momenti :
  - La dichiarazione (prototipo o *function prototype*)
    - L'interfaccia della funzione
    - Solitamente: prima del `main()`
  - La definizione
    - L'implementazione della funzione
    - Solitamente: al fondo del file, dopo il `main()`
  - La chiamata
    - L'utilizzo della funzione
    - Solitamente: dentro il corpo del `main()` o di altre funzioni

Dichiarazione o prototipo

```
int leggi(int min, int max) ;
```

- Tipo del valore di ritorno
- Nome della funzione
- Tipi e nomi dei parametri formali
- Punto-e-virgola

Scopo del prototipo

- Dichiarare che esiste una funzione con il nome dato, e dichiarare i tipi dei parametri formali e del valore di ritorno
- Dal prototipo in avanti, si può chiamare tale funzione dal corpo di qualsiasi funzione (compreso il `main`)
- Non è errore se la funzione non viene chiamata
- I file `.h` contengono centinaia di prototipi delle funzioni di libreria
- I prototipi sono posti solitamente ad inizio file

Definizione o implementazione

- Tipo del valore di ritorno
- Nome della funzione
- Tipi e nomi dei parametri formali

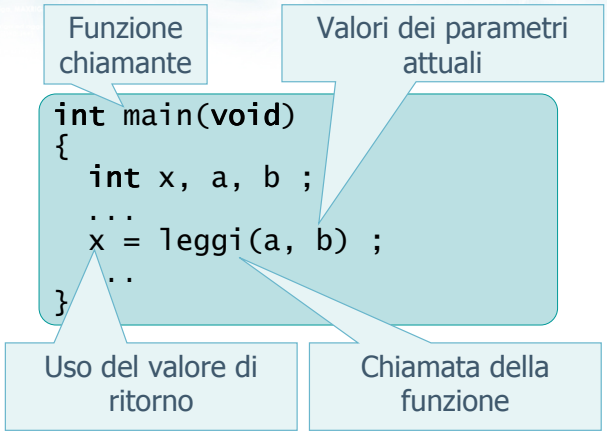
```
int leggi(int min, int max)
{
   . . . codice della funzione . . .
}
```

- Corpo della funzione { ... }
- Nessun punto-e-virgola

Scopo della definizione

- Contiene il codice vero e proprio della funzione
- Può essere posizionata ovunque nel file (al di fuori del corpo di altre funzioni)
- Il nome della funzione ed i tipi dei parametri e del valore di ritorno devono coincidere con quanto dichiarato nel prototipo
- Il corpo della funzione può essere arbitrariamente complesso, e si possono chiamare altre funzioni

Chiamata o invocazione



Meccanismo di chiamata

- Le espressioni corrispondenti ai parametri attuali vengono valutate (e ne viene calcolato il valore numerico)
  - Compaiono le variabili del chiamante
- I valori dei parametri attuali vengono copiati nei parametri formali
- La funzione viene eseguita
- All'istruzione `return`, il flusso di esecuzione torna al chiamante
- Il valore di ritorno viene usato o memorizzato

Riassumendo...

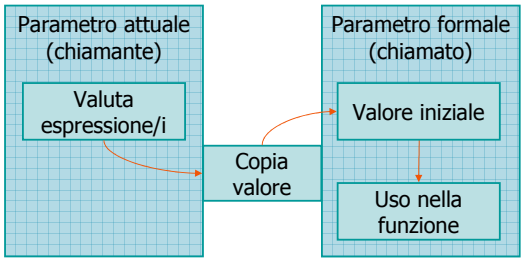
```
int leggi(int min, int max) ;

int main(void)
{
  int x, a, b ;
  ...
  x = leggi(a, b) ;
  ...
}

int leggi(int min, int max)
{
  ... codice della funzione ...
}
```


Funzioni in C

Passaggio dei parametri

Passaggio dei parametri

- Ogni volta che viene chiamata una funzione, avviene il trasferimento del valore corrente dei parametri attuali ai parametri formali



Conseguenza

- La funzione chiamata non ha assolutamente modo di
  - Conoscere il nome delle variabili utilizzate come parametri attuali
    - Ne conosce solo **il valore** corrente
  - Modificare il valore delle variabili utilizzate come parametri attuali
    - Riceve solamente una **copia** del valore
- Questo meccanismo è detto passaggio "by value" dei parametri
  - È l'unico possibile in C

Errore frequente

- Immaginare che una funzione possa modificare i valori delle variabili

```
void azzera(int x)
{
  x = 0 ;
}
```


Parametri di tipo vettoriale

- Il meccanismo di passaggio "by value" è chiaro nel caso di parametri di tipo scalare
- Nel caso di parametri di tipo array (vettore o matrice), il linguaggio C prevede che:
  - Un parametro di tipo array viene passato trasferendo una copia dell'indirizzo di memoria in cui si trova l'array specificato dal chiamante
  - Passaggio "by reference"

Conseguenza

- Nel passaggio di un vettore ad una funzione, il chiamato utilizzerà l'indirizzo a cui è memorizzato il vettore di partenza
- La funzione potrà quindi modificare il contenuto del vettore del chiamante
- Maggiori dettagli nella prossima lezione

Funzioni in C

Corpo della funzione

Variabili locali

- All'interno del corpo di una funzione è possibile definire delle **variabili locali**

```
int leggi(int min,
          int max)
{
  int v ;

  scanf("%d", &v) ;

  return v ;
}
```

Caratteristiche

- Le variabili locali sono accessibili solo dall'interno della funzione
- Le variabili locali sono indipendenti da eventuali variabili di ugual nome definite nel `main`
  - In ogni caso, dal corpo della funzione è impossibile accedere alle variabili definite nel `main`
- Le variabili locali devono avere nomi diversi dai parametri formali

Istruzioni eseguibili

- Il corpo di una funzione può contenere qualsiasi combinazione di istruzioni eseguibili
- Ricordare l'istruzione `return`

```
int leggi(int min,
          int max)
{
  int v ;

  scanf("%d", &v) ;

  return v ;
}
```

Funzioni

Parametri "by reference"

Parametri "by reference"

- Introduzione
- Operatori & e *
- Passaggio "by reference"
- Passaggio di vettori
- Esercizio "`strcpy`"

Parametri "by reference"

Introduzione

Passaggio dei parametri

- Il linguaggio C prevede il passaggio di parametri "by value"
  - Il chiamato non può modificare le variabili del chiamante
  - Il parametro formale viene inizializzato con una copia del valore del parametro attuale

Problemi

- Il passaggio "by value" risulta inefficiente qualora le quantità di dati da passare fossero notevoli
  - Nel caso del passaggio di vettori o matrici, il linguaggio C non permette il passaggio "by value", ma copia solamente l'indirizzo di partenza
  - Esempio: `strcmp`
- Talvolta è necessario o utile poter modificare il valore di una variabile nel chiamante
  - Occorre adottare un meccanismo per permettere tale modifica
  - Esempio: `scanf`

Soluzione

- La soluzione ad entrambi i problemi è la stessa:
  - Nel passaggio di vettori, ciò che viene passato è solamente l'indirizzo
  - Per permettere di modificare una variabile, se ne passa l'indirizzo, in modo che il chiamato possa modificare direttamente il suo contenuto in memoria
- Viene detto passaggio "by reference" dei parametri
  - Definizione impropria, in quanto gli indirizzi sono, a loro volta, passati "by value"

Parametri "by reference"

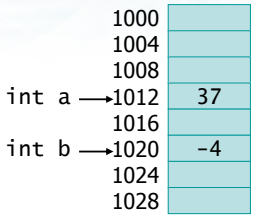
Operatori & e *

Operatori sugli indirizzi

- Per gestire il passaggio "by reference" dei parametri occorre
  - Conoscere l'indirizzo di memoria di una variabile
    - Operatore &
  - Accedere al contenuto di una variabile di cui si conosce l'indirizzo ma non il nome
    - Operatore *
- Prime nozioni della aritmetica degli indirizzi, che verrà approfondita in Unità successive

Operatore &

- L'operatore **indirizzo-di** restituisce l'indirizzo di memoria della variabile a cui viene applicato
  - &a è l'indirizzo 1012
  - &b è l'indirizzo 1020



Osservazioni

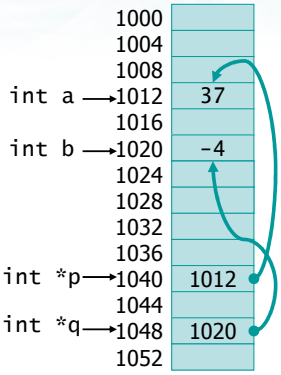
- L'indirizzo di una variabile viene deciso dal compilatore
- L'operatore & si può applicare solo a variabili singole, non ad espressioni
  - Non ha senso &(a+b)
  - Non ha senso &(3)
- Conoscere l'indirizzo di una variabile permette di leggerne o modificarne il valore senza conoscerne il nome

Variabili "puntatore"

- Per memorizzare gli indirizzi di memoria, occorre definire opportune variabili di tipo "indirizzo di..."
- Nel linguaggio C si chiamano **puntatori**
- Un puntatore si definisce con il simbolo *
  - `int *p ; /* puntatore ad un valore intero */`
  - `float *q ; /* puntatore ad un valore reale */`

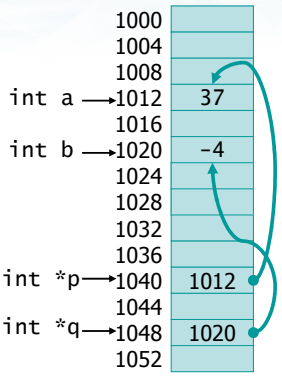
Esempio

```
int main(void)
{
  int a, b ;
  int *p, *q ;

  a = 37 ;
  b = -4 ;

  p = &a ;
     /* p "punta a" a */

  q = &b ;
     /* q "punta a" b */

}
```



Operatore *

- L'operatore di **accesso indiretto** permette di accedere, in lettura o scrittura, al valore di una variabile di cui si conosce l'indirizzo
  - `*p` equivale ad a
  - `*q` equivale a b
  - `*p = 0 ;`
  - `if( *q > 0 ) ...`



Costrutti frequenti

Costrutto	Significato
`int x ;`	x è una variabile intera
`int *p ;`	p è un puntatore a variabili intere
`p = &x ;`	p punta ad x
`*p = 0 ;`	Azzera la variabile puntata da p (cioè x)
`b = *p ;`	Leggi il contenuto della variabile puntata da p e copialo in b

Parametri "by reference"

Passaggio "by reference"

Passaggio "by reference"

- Obiettivo: passare ad una funzione una variabile, in modo tale che la funzione la possa modificare
- Soluzione:
  - Definire un parametro attuale di tipo puntatore
  - Al momento della chiamata, passare l'indirizzo della variabile (anziché il suo valore)
  - All'interno del corpo della funzione, fare sempre accesso indiretto alla variabile di cui è noto l'indirizzo

Esempio: "Azzera"

- Scrivere una funzione `azzera`, che riceve un parametro di tipo intero (by reference) e che azzera il valore di tale parametro

Soluzione

```
void azzera( int *v ) ;
```

```
int main( void )
{
  int x ;
  ...
  azzera(&x) ;
  ...
}
```

```
void azzera( int *v )
{
  *v = 0 ;
}
```

Esempio: "Scambia"

- Scrivere una funzione `scambia`, che riceve due parametri di tipo intero (by reference) e che scambia tra di loro i valori in essi contenuti

Soluzione

```
void scambia( int *p, int *q ) ;
```

```
int main( void )
{
  int a,b ;
  ...
  scambia(&a, &b) ;
  ...
}
```

```
void scambia( int *p, int *q )
{
  int t ;
  t = *p ;
  *p = *q ;
  *q = t ;
}
```


Osservazione

- Il meccanismo di passaggio by reference spiega (finalmente!) il motivo per cui nella funzione `scanf` è necessario specificare il carattere & nelle variabili lette
- Le variabili vengono passate by reference alla funzione `scanf`, in modo che questa possa scrivervi dentro il valore immesso dall'utente

Parametri "by reference"

Passaggio di vettori

Passaggio di vettori e matrici

- Nel linguaggio C, il nome di un array (vettore o matrice) è automaticamente sinonimo del puntatore al suo primo elemento

```
int main(void)
{
  int v[10] ;
  int *p ;

  p = & v[0] ;

}
```

p e v sono del tutto equivalenti

Conseguenze

- Quando il parametro di una funzione è di tipo array (vettore o matrice)
  - L'array viene passato direttamente "by reference"
  - Non è necessario l'operatore & per determinare l'indirizzo
    - È sufficiente il nome del vettore
  - Non è necessario l'operatore * per accedere al contenuto
    - È sufficiente l'operatore di indicizzazione []
  - Non è possibile, neppure volendolo, passare un array "by value"

Esercizio "Duplicati"

- Scrivere una funzione che, ricevendo due parametri
  - Un vettore di `double`
  - Un intero che indica l'occupazione effettiva di tale vettore

  possa determinare se vi siano valori duplicati in tale vettore
- La funzione ritornerà un intero pari a 1 nel caso in cui vi siano duplicati, pari a 0 nel caso in cui non ve ne siano

Soluzione (1/3)

```
int duplicati(double v[], int N) ;
/*
Riceve in ingresso il vettore v[] di double
che contiene N elementi (da v[0] a v[N-1])

Restituisce 0 se in v[] non vi sono duplicati
Restituisce 1 se in v[] vi sono duplicati

Il vettore v[] non viene modificato
*/
```


Soluzione (2/3)

```
int duplicati(double v[], int N)
{
  int i, j ;

  for(i=0; i<N; i++)
  {
    for(j=i+1; j<N; j++)
    {
      if(v[i]==v[j])
        return 1 ;
    }
  }
  return 0 ;
}
```


Soluzione (3/3)

```
int main(void)
{
  const int MAX = 100 ;
  double dati[MAX] ;
  int Ndati ;
  int dupl ;

  ...
  dupl = duplicati(dati, Ndati) ;
  ...
}
```


Errore frequente

- Nel passaggio di un vettore occorre indicarne solo il nome

```
dupl = duplicati(dati, Ndati) ;
```

~~`dupl = duplicati(dati[], Ndati) ;`~~

~~`dupl = duplicati(dati[MAX], Ndati) ;`~~

~~`dupl = duplicati(dati[Ndati], Ndati) ;`~~

~~`dupl = duplicati(&dati, Ndati) ;`~~

Osservazione

- Nel caso dei vettori, il linguaggio C permette solamente il passaggio by reference
  - Ciò significa che il chiamato ha la possibilità di modificare il contenuto del vettore
- Non è detto che il chiamato effettivamente ne modifichi il contenuto
  - La funzione duplicati analizza il vettore senza modificarlo
  - Esplicitarlo sempre nei commenti di documentazione

Parametri "by reference"

Esercizio "strcpy"

Esercizio "strcpy"

- Si implementi, sotto forma di funzione, la ben nota funzione di libreria `strcpy` per la copia di due stringhe

Soluzione (1/2)

```
void strcpy(char *dst, char *src) ;
/*
Copia il contenuto della stringa src
nella stringa dst

Assume che src sia 0-terminata, e restituisce
dst in forma 0-terminata.

Assume che nella stringa dst vi sia spazio
sufficiente per la copia.

La stringa src non viene modificata.
*/
```


Soluzione (2/2)

```
void strcpy(char *dst, char *src)
{
  int i ;

  for(i=0; src[i]!=0; i++)
  {
    dst[i] = src[i] ;
  }
  dst[i] = 0 ;
}
```


Osservazione

- La funzione può essere dichiarata in due modi:
  - `void strcpy(char *dst, char *src)`
  - `void strcpy(char dst[], char src[])`
- Sono forme assolutamente equivalenti
- Tutte le funzioni di libreria che lavorano sulle stringhe accettano dei parametri di tipo `char *`

Funzioni

La funzione `main()`

La funzione `main()`

- Interfaccia con il sistema operativo
- Argomenti sulla linea di comando
- Parametri `argc` e `argv`
- Valore di ritorno del programma
- La funzione `exit`
- Esercizio "Calcolatrice"

La funzione `main()`

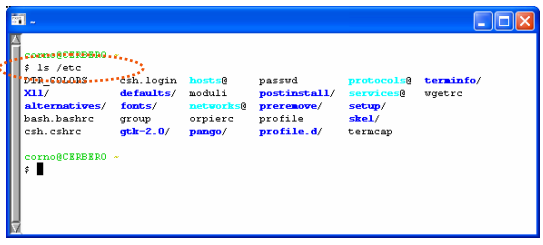
Interfaccia con il sistema operativo

La funzione `main()`

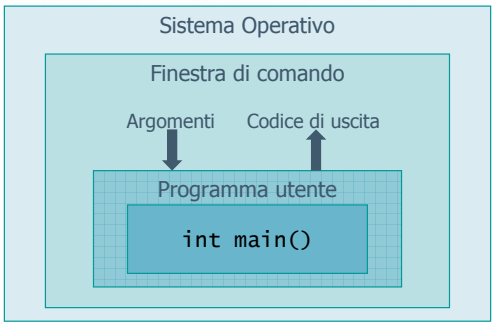
- La funzione `main()`, presente in tutti i programmi C, è una funzione come tutte le altre
- Unica particolarità: viene chiamata automaticamente dal Sistema Operativo, appena il programma viene avviato
  - Non esiste mai una chiamata esplicita a `main()`
  - L'interfaccia della funzione viene definita dalle caratteristiche del sistema operativo

Interfaccia in modalità "console"

- Ricordiamo che il linguaggio C si è evoluto con interfacce "a caratteri"
- L'attivazione di un programma avviene digitandone il nome in una finestra di comando



Il modello "console"



Interfaccia del programma

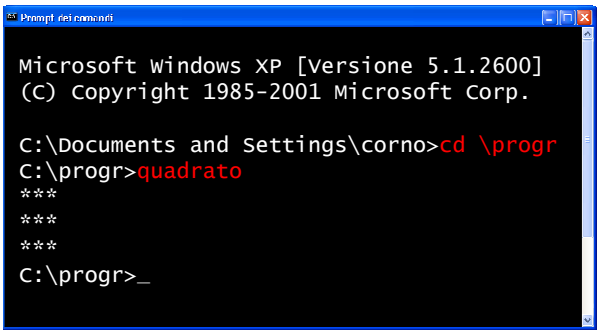
- La finestra di comando permette di passare al programma una serie di **argomenti**
  - Zero, una o più stringhe di testo
  - Utilizzate dal programma come dati in ingresso
- Al termine dell'esecuzione, il programma restituisce un **codice di uscita**
  - Numero intero
  - Indica eventuali condizioni di errore
- Durante l'esecuzione, il programma può accedere all'input (tastiera) e all'output (schermo)

La funzione `main()`

Argomenti sulla linea di comando

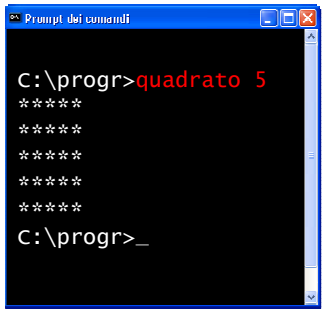
La linea di comando (1/2)

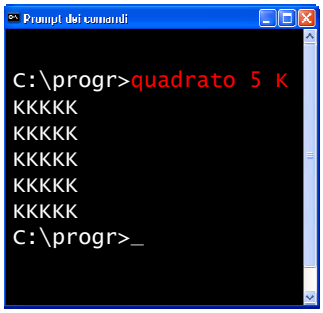
- L'attivazione di un programma avviene digitando il suo nome in una finestra di comando



La linea di comando (2/2)

- È possibile passare dei parametri all'attivazione del programma





Nomenclatura

- Parametri sulla linea di comando
  - Command line parameters
- Argomenti del programma
  - Program arguments
- Parametri di attivazione
- Parametri del main
- Argomenti del main
- Opzioni [sulla linea di comando]

Caratteristiche

- Numero variabile di parametri
  - Anche nessuno
- Tipo variabile
  - Numeri
  - Caratteri o Stringhe
- Il chiamante (sistema operativo) non ha modo di sapere quanti parametri servono al programma né di che tipo
  - Verranno trattati in modo standardizzato

Standardizzazione dei parametri

- Gli argomenti sulla linea di comando vengono trattati come un **vettore di stringhe**
- Il programma riceve
  - Una copia del vettore di stringhe
  - Un valore numerico che indica quante stringhe sono presenti

Esempi

```
C:\progr>quadrato
```

- Numero argomenti = 0

```
C:\progr>quadrato 5
```

- Numero argomenti = 1
- Argomento 1 = "5"

```
C:\progr>quadrato 5 K
```

- Numero argomenti = 2
- Argomento 1 = "5"
- Argomento 2 = "K"

La funzione `main()`

Parametri `argc` e `argv`

Parametri formali del `main`

- I parametri sulla linea di comando sono disponibili al `main` attraverso i suoi parametri formali
- La definizione completa della funzione `main` è:

```
int main(int argc, char *argv[]) ;
```

Argument count

Argument values

Il parametro `argc`

- `int argc`
- Numero di parametri sulla linea di comando
  - Incrementato di uno, in quanto il nome del programma viene considerato come un parametro
- Se non vi sono argomenti effettivi, vale 1
- Se vi sono k argomenti effettivi, vale k+1

Il parametro `argv`

- `char *argv[]`
- È un vettore di stringhe
- Ogni stringa è un parametro del programma
- Vi sono `argc` diverse stringhe
- La prima stringa, `argv[0]`, è il nome del programma
- La seconda stringa, `argv[1]`, è il primo argomento (se esiste)
- …
- L'ultimo argomento è in `argv[argc-1]`

Esempi

```
C:\progr>quadrato
```

- argc==1
- argv[0]=="quadrato"

```
C:\progr>quadrato 5
```

- argc==2
- argv[0]=="quadrato"
- argv[1]=="5"

```
C:\progr>quadrato 5 K
```

- argc==3
- argv[0]=="quadrato"
- argv[1]=="5"
- argv[2]=="K"

Per capire meglio...

arg.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i ;

    printf("argc = %d\n", argc) ;
    for(i=0; i<argc; i++)
    {
        printf("argv[%d] = \"%s\"\n",
            i, argv[i]) ;
    }
}
```


Osservazione

- Il vettore `argv` contiene i dati sotto forma esclusivamente di stringa
- Qualora uno dei dati richiesti sia di tipo numerico (`int` o `double`), occorre effettuare la conversione da stringa a numero
  - `i = atoi(argv[1]) ;`
  - `r = atof(argv[1]) ;`
- Se il parametro è invece una stringa, conviene copiarlo in una variabile mediante `strcpy`

La funzione `main()`

Valore di ritorno del programma

Valore di ritorno

- Al termine dell'elaborazione il programma restituisce un numero intero al sistema operativo
- Tale valore viene spesso ignorato, ma in caso di esecuzione "batch" è possibile interrogarlo a livello di sistema operativo
  - in MS-DOS, tramite la variabile ERRORLEVEL
    - `echo %errorlevel%`
  - in sistemi Unix, mediante la macro $?
    - `echo $?`

Convenzioni

- Il valore di ritorno è un int, ma per compatibilità si preferisce ritornare degli "interi positivi piccoli"
- Convenzionalmente
  - Il valore di ritorno pari a 0 indica "programma terminato correttamente"
  - Il valore di ritorno diverso da 0 indica "programma terminato anormalmente a causa di un errore"
    - Il valore specifico ritornato (1, 2, 3, ...) può indicare la causa dell'errore

Esempio

La funzione `main()`

La funzione `exit`

Restituzione del valore di ritorno

- Quando il programma termina, deve restituire il valore di ritorno
  - ==0, se tutto OK
  - !=0, se errore
- Il modo più semplice per restituire tale valore è di utilizzare l'istruzione `return` all'interno della funzione `main`
  - L'elaborazione viene immediatamente interrotta
  - Il valore ritornato viene passato al sistema operativo

Esempio

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   . . .
   . . .
   . . .
   return 0 ;
}
```


La funzione `exit`

- Esiste inoltre la funzione di libreria `exit`, dichiarata in `<stdlib.h>`, che assolve alla stessa funzione
  - Interrompe l'esecuzione del programma
  - Ritorna il valore specificato
- Il vantaggio rispetto all'istruzione `return` è che può essere usata all'interno di qualsiasi funzione, non solo del `main`

```
void exit(int value) ;
```


Esempio

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   . . .
   . . .
   . . .
   exit(0) ;
}
```

Suggerimento

- Ricordare sempre di ritornare un valore
- Mettere come ultima istruzione del `main`: `exit(0);`
- Per eventuali condizioni di errore (parametri assenti, valori illegali, ...) che non possano essere corrette dal programma, restituire un valore positivo: `exit(1) ;`
  - Tali errori possono essere controllati dall'interno di qualsiasi funzione: la `exit` interrompe comunque l'intero programma

La funzione `main()`

Esercizio "Calcolatrice"

Esercizio "Calcolatrice"

- Si scriva un programma da utilizzarsi come semplice calcolatrice sulla linea di comando
- Il programma, denominato `calcola`, accetta 3 parametri sulla linea di comando
  - Il primo ed il terzo parametro sono degli operandi, espressi come numeri reali
  - Il secondo parametro è un operatore, scelto tra +, -, * e /
- Il programma stampa il risultato corrispondente all'operazione

Analisi

```
Prompt dei comandi
c:\progr>calcola 3 + 2
Risultato: 5.0000
c:\progr>calcola 3 * 2
Risultato: 6.0000
c:\progr>calcola 3 $ 2
Errore: operatore non riconosciuto
c:\progr>calcola 3 +
Errore: operando mancante
```


Soluzione (1/5)

calcola.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    double v1, v2 ;
    char op[20] ;
```


Soluzione (2/5)

calcola.c

```
if(argc!=4)
{
    printf("Errore: numero parametri
            insufficiente\n");
    exit(1) ;
}

v1 = atof(argv[1]) ;
strcpy(op, argv[2]) ;
v2 = atof(argv[3]) ;
```

Soluzione (3/5)

calcola.c

```
    if(strcmp(op, "+")==0)
        printf("Risultato: %f\n", v1 + v2 ) ;
    else if(strcmp(op, "-")==0)
        printf("Risultato: %f\n", v1 - v2 ) ;
    else if(strcmp(op, "*")==0)
        printf("Risultato: %f\n", v1 * v2 ) ;
```


Soluzione (4/5)

calcola.c

```
    else if(strcmp(op, "/")==0)
    {
        if(v2==0)
        {
            printf("Errore: divisione per zero\n");
            exit(2) ;
        }
        else
            printf("Risultato: %f\n", v1 / v2 ) ;
    }
```


Soluzione (5/5)

calcola.c

```
    else
    {
        printf("Errore: operatore
                non riconosciuto\n") ;
        exit(3) ;
    }
    exit(0) ;
}
```

Funzioni

Esercizi proposti

Esercizi proposti

- Esercizio "Confronto tra date"
- Esercizio "Quadrato"
- Esercizio "Indovina numero"

Esercizi proposti

Esercizio "Confronto tra date"

Esercizio "Confronto tra date"

- Si scriva un programma che chieda all'utente di inserire due date (giorno, mese, anno) e determini:
  - Se le date solo uguali
  - Se la prima data **precede** la seconda
  - Se la prima data **segue** la seconda
- Il programma dovrà porre particolare attenzione a **non accettare** date non valide
  - Esempio: 30/02/1984
  - Esempio: 10/14/2001

Analisi

```
Prompt dei comandi
Confronto tra date

Inserisci la PRIMA data
Giorno: 15
Mese: 3
Anno: 2007
Inserisci la SECONDA data
Giorno: 26
Mese: 4
Anno: 1967

La prima data 15/3/2007 SEGUE
la seconda data 26/4/1967
```


Controlli

- $1 \leq$ giorno ≤ 31
- $1 \leq$ mese ≤ 12
- $1900 <$ anno < 2100
- Se mese = 4, 6, 9, 11, allora giorno ≤ 30
- Se mese = 2, allora giorno ≤ 29
- Se mese = 2 e l'anno non è bisestile, allora giorno ≤ 28
  - L'anno è bisestile se è multiplo di 4

Soluzione

- Scrivere una funzione per la lettura della data, che comprenda al suo interno tutti i controlli:

```
void leggidata( int *giorno,
      int *mese, int *anno ) ;
```

- La funzione restituisce, nelle 3 variabili passate *by reference*, le componenti (giorno, mese, anno) della data
- La funzione **garantisce** che la data restituita è **corretta**

Programma principale

date.c

```
int main(void)
{
    int g1, m1, a1 ;
    int g2, m2, a2 ;

    printf("Confronto tra date\n\n") ;

    printf("Inserisci la PRIMA data\n") ;
    leggidata( &g1, &m1, &a1 ) ;

    printf("Inserisci la SECONDA data\n") ;
    leggidata( &g2, &m2, &a2 ) ;

    /* Confronto delle date */

} /* main */
```

Confronto delle date

date.c

```
if( g1 == g2 && m1 == m2 && a1 == a2 )
    printf("Le date sono uguali\n") ;
else if( a1<a2 ||
         (a1==a2 && m1<m2) ||
         (a1==a2 && m1==m2 && g1<g2) )
{
    printf("La prima data %d/%d/%d "
     "PRECEDE la seconda %d/%d/%d\n",
        g1, m1, a1, g2, m2, a2) ;
}
else
{
    printf("La prima data %d/%d/%d "
     "SEGUE la seconda %d/%d/%d\n",
        g1, m1, a1, g2, m2, a2) ;
}
```


Funzione leggidata (1/5)

date.c

```
void leggidata( int *giorno, int *mese,
                int *anno )
{
  int g, m, a ;
  int ok ;

  do {
    do {
      printf("Giorno: ") ;
      scanf("%d", &g) ;

      if(g<1 || g>31)
        printf("Giorno non valido\n");
    } while( g<1 || g>31 ) ;
```


Funzione leggidata (2/5)

date.c

```
    do {
      printf("Mese: ") ;
      scanf("%d", &m) ;

      if(m<1 || m>12)
        printf("Mese non valido\n");
    } while( m<1 || m>12 ) ;

    do {
      printf("Anno: ") ;
      scanf("%d", &a) ;

      if ( a<=1900 || a>=2100 )
        printf("Anno non valido\n") ;
    } while( a<=1900 || a>=2100 ) ;
```


Funzione leggidata (3/5)

date.c

```
    ok = 1 ;
    if( g>30 && (m==4 || m==6 ||
        m==9 || m==11) )
    {
      ok = 0 ;
      printf("Il mese %d non "
        "ha %d giorni\n", m, g);
    }
    else if ( g>29 && m==2 )
    {
      ok = 0 ;
      printf("Il mese %d non "
        "ha %d giorni\n", m, g);
    }
```

Funzione leggidata (4/5)

date.c

```
else if ( g==29 && m==2 && a%4!=0 )
    {
      ok = 0 ;
      printf("Il mese %d non "
         "ha %d giorni perche' "
         "l'anno %d non e' bisestile\n",
         m, g, a);
    }
```

Funzione leggidata (5/5)

date.c

```
  } while( ok==0 ) ;

  *giorno = g ;
  *mese = m ;
  *anno = a ;

  return ;

} /* leggidata */
```


Esercizi proposti

Esercizio "Quadrato"

Esercizio "Quadrato" (1/2)

- Si scriva un programma, denominato `quadrato`, che stampi a video un quadrato composto di caratteri tutti uguali.
- Il programma riceve due argomenti sulla linea di comando:
  - Il primo argomento indica la **dimensione** del quadrato (ossia il numero di righe e colonne di cui è composto)
  - Il secondo argomento indica il **carattere** di cui è composto il quadrato

Esercizio "Quadrato" (2/2)

- Gli argomenti sono opzionali: se il carattere viene omesso, occorre stampare "*". Se anche la dimensione viene omessa, si assuma pari a 3

Analisi

```
C:\progr>quadrato
```

- Quadrato 3x3 di "*"

```
C:\progr>quadrato 5
```

- Quadrato 5x5 di "*"

```
C:\progr>quadrato 5 K
```

- Quadrato 5x5 di "K"

Dal punto di vista del main

```
C:\progr>quadrato
```

- argc==1
- argv[0]=="quadrato"

```
C:\progr>quadrato 5
```

- argc==2
- argv[0]=="quadrato"
- argv[1]=="5"

```
C:\progr>quadrato 5 K
```

- argc==3
- argv[0]=="quadrato"
- argv[1]=="5"
- argv[2]=="K"

Soluzione (1/4)

quadrato.c

```
int main(int argc, char *argv[])
{
    int dim ;
    char ch ;
    int i, j ;

    if (argc==1)
    {
        dim = 3 ;
        ch = '*' ;
    }
```


Soluzione (2/4)

quadrato.c

```
    else if (argc==2)
    {
       dim = atoi(argv[1]) ;
       if( dim<1 || dim>20 )
       {
         printf("Dimens. non valida\n") ;
         exit(1) ;
       }
       ch = '*' ;
    }
```

Soluzione (3/4)

quadrato.c

```
    else if (argc==3)
    {
        dim = atoi(argv[1]) ;
        if( dim<1 || dim>20 )
        {
          printf("Dimens. non valida\n");
          exit(1) ;
        }
        ch = argv[2][0] ;
        if(strlen(argv[2])!=1)
        {
          printf("Carattere non valido\n");
          exit(1) ;
        }
    }
```

Soluzione (4/4)

quadrato.c

```
    else
    {
        printf("Numero argomenti "
               "non valido\n") ;
        exit(1) ;
    }

    for(i=0; i<dim; i++)
    {
        for(j=0; j<dim; j++)
            putchar(ch) ;
        putchar('\n') ;
    }

    exit(0) ;
}
```


Esercizi proposti

Esercizio "Indovina numero"

Esercizio "Indovina numero"

- Si realizzi un programma in C per permettere a due giocatori umani di giocare ad "indovina il numero"
- Il primo giocatore attiva il programma, denominato `segreto`, passandogli sulla linea di comando un numero intero tra 1 e 100
- Il secondo giocatore farà una serie di tentativi, immettendoli via tastiera
- Ad ogni tentativo il programma dirà se il numero tentato è più alto o più basso del numero da indovinare

Soluzione (1/3)

segreto.c

```
int main(int argc, char *argv[])
{
    int segreto ;
    int cont, tent ;

    /* Acquisisci dalla linea
     di comando il numero "segreto" */
    if( argc != 2 )
    {
        printf("Numero di parametri "
               "errato\n") ;
        exit(1) ;
    }
```

Soluzione (2/3)

segreto.c

```
segreto = atoi( argv[1] ) ;

if( segreto<1 || segreto>100 )
{
    printf("Numero segreto "
           "errato\n") ;
    exit(1) ;
}

printf("INDOVINA IL NUMERO\n\n");
```

Soluzione (3/3)

segreto.c

```
    cont = 1 ;
    do {
        printf("Tentativo %d: ", cont );
        scanf("%d", &tent) ;

        if( tent < segreto )
            printf("Basso...\n") ;
        else if (tent > segreto )
            printf("Alto...\n") ;

        cont++ ;
    } while (tent != segreto) ;
    cont-- ;

    printf("Indovinato: %d tentativi\n",
           cont) ;
```

Funzioni

Sommario

Argomenti trattati

- Definizione di funzioni in C
  - Prototipo
  - Implementazione
- Passaggio di parametri alle funzioni
  - By value
  - By reference
  - Vettori
- Interfaccia del `main`
  - Parametri sulla linea di comando
  - Valore di ritorno

Tecniche di programmazione

- Decomporre il programma in più funzioni, implementando ciascuna individualmente
- Identificare le parti ripetitive e racchiuderle in apposite funzioni
- Permette il passaggio di parametri al programma
- Analizzare i parametri passati dall'utente

Materiale aggiuntivo

- Sul CD-ROM
  - Testi e soluzioni degli esercizi trattati nei lucidi
  - Scheda sintetica
  - Esercizi risolti
  - Esercizi proposti
- Esercizi proposti da altri libri di testo