



Programmazione I

Il Linguaggio C

Le Funzioni

Daniel Riccio

Università di Napoli, Federico II

24 novembre 2021



Sommario



- Argomenti
 - Puntatori a funzione
 - La funzione main
 - Esercizi

Indirizzo di una funzione



Si può chiamare una funzione utilizzando l'indirizzo di memoria dal quale inizia il codice eseguibile della funzione stessa

L'indirizzo di memoria di una funzione può essere assegnato ad una variabile puntatore, memorizzato in un vettore, passato a una funzione, restituito da una funzione

Definizione di variabili



```
tipo (*nome) (parametri) ;
```

definisce la variabile **nome** come puntatore a una funzione che restituisce un valore del **tipo** indicato e richiede i **parametri** indicati

Le parentesi intorno al nome sono necessarie altrimenti si ha:

```
tipo *nome (parametri) ;
```

e questa costituisce il prototipo di una funzione che restituisce un puntatore del **tipo** indicato e richiede i **parametri** indicati

Definizione di variabili



```
double (*fp) ();
```

definisce **fp** come variabile puntatore a una funzione che restituisce un **double**, nulla è indicato per i suoi parametri, quindi non vengono controllati

```
double (*fp) (double, double);
```

definisce **fp** come variabile puntatore a una funzione che ha come parametri due **double** e restituisce un **double**

```
int (*fpv[3]) (long);
```

definisce **fpv** come vettore di 3 elementi, ciascuno è un puntatore a una funzione che ha come parametri un **long** e restituisce un **int**

Le definizioni esterne inizializzano a **NULL** le variabili puntatori a funzioni

Definizione di variabili

Il nome di una funzione equivale al suo indirizzo di memoria (come per i vettori)

Con le seguenti funzioni (prototipi):

```
double pow(double, double);  
double atan2(double, double);  
int f1(long), f2(long);
```

si possono avere le seguenti inizializzazioni:

```
double (*fp)(double, double) = NULL;  
double (*fp)(double, double) = pow;  
int (*fpv[3])(long) = {f1, f2}; /*+1 NULL*/  
int (*fpv[3])(long) = {NULL}; /*3 NULL*/
```

Assegnazione e confronto



Il nome di una funzione equivale al suo indirizzo di memoria (come per i vettori)

L'operatore **&** è ininfluente se applicato al nome di una funzione

Esempi (l'operatore **&** può essere omesso):

```
fp = &atan2;  
fp = atan2;  
fpv[2] = &f1;  
fpv[2] = f1;
```

Il confronto fra puntatori a funzione è un normale confronto tra puntatori, permette ad esempio di determinare se il puntatore si riferisce ad una certa funzione o no oppure se è `NULL`

```
if (fp == pow)  
printf("fp punta a pow\n");
```

Chiamata della funzione

La funzione il cui puntatore è stato memorizzato nella variabile `fp` può essere richiamata in due modi equivalenti:

```
(*fp) (argomenti)
```

```
fp (argomenti)
```

Il primo metodo rende più evidente che si tratta di un puntatore a funzione e non del nome di una funzione, il secondo è più leggibile e comodo

```
x = (*fp) (2.23, 4.76);  
x = fp (2.23, 4.76);  
i = (*fpv[1]) (22);  
i = fpv[1] (22);
```

Ricordando che:

```
double (*fp) (double, double) = pow;  
int f1 (long);  
int (*fpv[3]) (long) = { f1, f2 };
```


Passaggio a funzione

Un puntatore a funzione può essere passato ad una funzione come argomento

Questo permette di passare ad una funzione [puntatori a] funzioni diverse per ottenere dalla stessa funzione comportamenti diversi

Il parametro attuale deve essere il nome di una funzione

Il parametro formale deve essere il prototipo delle funzioni chiamabili (è lì che viene definito il nome del puntatore visto all'interno della funzione stessa)

Passaggio a funzione



Esempio:

Si supponga di avere le due funzioni seguenti:

`int piu(int a,int b)` effettua la somma di 2 numeri interi
`int meno(int a,int b)` effettua la differenza di 2 numeri interi

```
int piu(int a,int b) {return a+b;}  
int meno(int a,int b) {return a-b;}  
int calc(int x,int y, int (*funz)(int,int))  
{  
return (*funz)(x,y);  
}
```

```
int main() {  
    int m, n;  
    m = calc(12, 23, piu);    //calcola la somma dei due valori  
    n = calc(12, 23, meno);  //calcola la differenza dei due valori  
    ...  
}
```

Valore restituito da una funzione



Una funzione può restituire un puntatore a funzione

Si definisce con l'istruzione **typedef** il tipo del puntatore-a-funzione restituito dalla funzione chiamata: **typedef int (*TipoFunz) (int,int) ;**

TipoFunz è un nuovo tipo: puntatore-a-funzione-che-ha-2-argomenti-**int**-e-restituisce-un-**int**

E lo si usa nel solito modo: **TipoFunz operaz (int x) ;**

```
typedef int (*TipoFunz) (int, int);
int piu(int a,int b) {return a+b;}
int meno(int a,int b) {return a-b;}
TipoFunz operaz(int f)
{
    static TipoFunz fpv[2]={piu,meno};
    return fpv[f];
}
main()
{
    int y;
    TipoFunz op; /* var. puntatore */
    op = operaz(1); /* scelta op. */
    y = (*op) (12, 23);
}
```

La funzione main



La funzione **main()**, presente in tutti i programmi C, è una funzione come tutte le altre

Unica particolarità: viene chiamata dal **Sistema Operativo**, appena il programma viene avviato

- Non esiste mai una chiamata a **main()**
- L'interfaccia della funzione viene definita dalle caratteristiche del sistema operativo

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

```
Prompt dei comandi

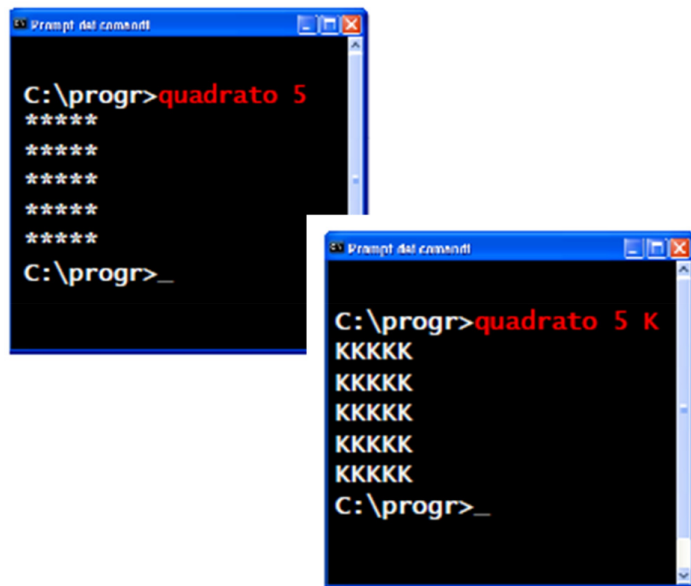
Microsoft Windows XP [Versione 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\corno>cd \progr
C:\progr>quadrato
***
***
***
C:\progr>_
```

La funzione main

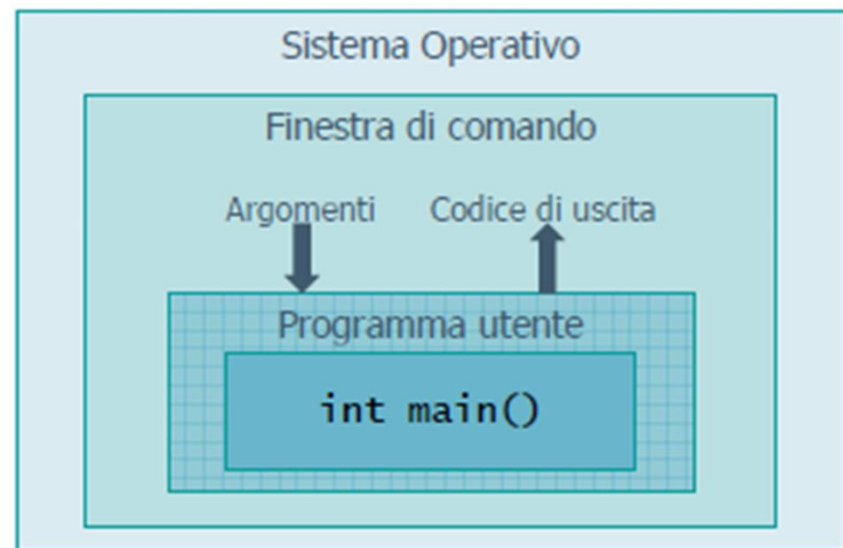
La finestra di comando permette di passare al programma una serie di argomenti

- Nessuna, una o più stringhe di testo
- Utilizzate dal programma come dati in ingresso



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

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



La funzione main



Numero variabile di argomenti

- Anche nessuno

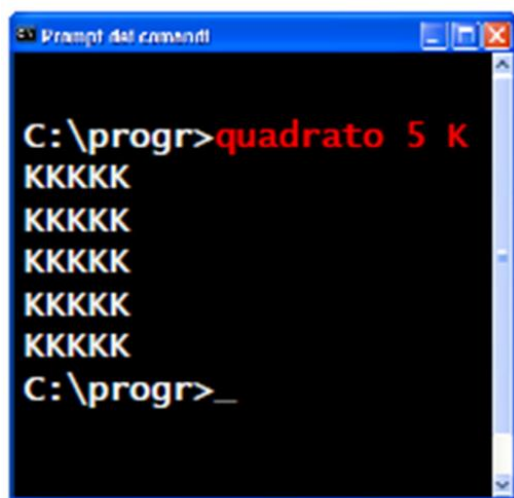
Tipo argomenti

- Numeri
- Caratteri o stringhe

Gli argomenti sono anche indicati come
«argomenti del programma»
«argomenti del main»
«parametri del programma»
«parametri sulla linea di comando»

Il chiamante (sistema operativo) non ha modo di sapere quanti parametri servono al programma, né di che tipo

Verranno trattati in modo standardizzato



```
Prompt dei comandi

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

La funzione main



Gli argomenti sulla linea di comando vengono trattati come un vettore di stringhe

Anche nessuno

Il programma riceve

Una copia del vettore di stringhe

Un valore numerico che indica quante stringhe sono presenti

```
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"

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

Argument count

Argument values

La funzione main



■ `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, vale 1
 - Se vi sono k argomenti, vale $k+1$

■ `char *argv`

■ E' 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]`

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

Argument count

Argument values

```
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"
```

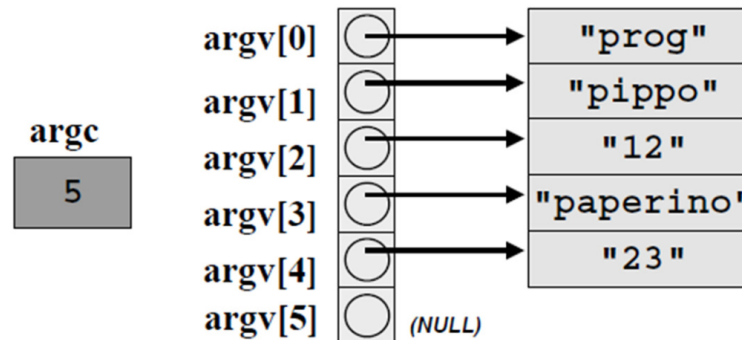

La funzione main

- Il vettore **argv** contiene i dati sotto forma di stringa
- Per i dati di tipo numerico (**int** o **double**) occorre effettuare la conversione da stringa a numero
 - `i=atoi(argv[1]);`
 - `r=atof(argv[1]);`

```
C:\progr>progr pippo 12 paperino 23
```

Se il parametro è invece una stringa, è possibile

- Utilizzare un puntatore alla stringa di interesse
- Copiarlo in una variabile (all'occorrenza, mediante **strcpy**)



```
#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]) ;
    }
}
```

La funzione main



- Scrivere un programma che dato un carattere e una stringa da linea di comando, conti quante volte il carattere compare nella stringa

- `c:> contaOccorrenze c stringaincui cercare`

■ Algoritmo

- Recuperare gli argomenti
 - Scandire la stringa carattere per carattere, contando le occorrenze del carattere dato
 - Scrivere il numero di occorrenze a video

La funzione main

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

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

```
{
```

```
    int cont=0, i=0;
```

```
    char ch=argv[1][0];
```

```
    char *s=argv[2];
```

```
    while (s[i] != '\0')
```

```
        if (s[i++] == ch)
```

```
            cont++;
```

```
    printf ("totale occorrenze: %d", cont);
```

```
}
```

La funzione main



- Scrivere un programma che sommi tutti i numeri (supposti interi) passati come argomenti a linea di comando e stampi a video questo risultato
 - `c:> sommatutto 12 34 5 67`

■ Algoritmo

- Recuperare gli argomenti
- Trasformare ciascun argomento (stringa) in un intero: si utilizza una funzione di libreria `atoi(char *)`
- Scrivere il risultato a video

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

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

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

```
{
```

```
    int sum=0, i=0;
```

```
    while (i < (argc-1))
```

```
        sum += atoi(argv[++i]);
```

```
    printf ("somma totale : %d", sum);
```

```
}
```

La funzione main



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

int main(int argc, char *argv[])
{
    int sum=0, i=0;

    while(i < (argc-1))
        sum += atoi(argv[++i]);

    printf ("somma totale : %d", sum);
}
```

Il Preprocessore



Modifica il codice C prima che venga eseguita la traduzione vera e propria

Le direttive al preprocessore riguardano:

- inclusione di file (**#include**)
- definizione di simboli (**#define**)
- sostituzione di simboli (**#define**)
- compilazione condizionale (**#if**)
- macroistruzioni con parametri (**#define**)

Non essendo istruzioni C non richiedono il ';' finale

Il Preprocessore



La riga con **#include** viene sostituita dal contenuto testuale del file indicato

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

Il nome del file può essere completo di percorso

Ha due forme:

- **#include** <file.h> Il file viene cercato nelle directory del compilatore
- **#include** "file.h" Il file viene cercato prima nella directory dove si trova il file C e poi, se non trovato, nelle directory del compilatore

I file inclusi possono a loro volta contenere altre direttive **#include**

La direttiva **#include** viene in genere collocata in testa al file sorgente C, prima della prima funzione

Generalmente, i file inclusi non contengono codice eseguibile, ma solo dichiarazioni (es. prototipi e variabili **extern**) e altre direttive

Il Preprocessore



#define nome

definisce il simbolo denominato **nome**

#define DEBUG

I simboli vengono utilizzati dalle altre direttive del preprocessore (ad es. si può verificare se un simbolo è stato definito o no con **#if**)

Lo scope di **nome** si estende dalla riga con la definizione fino alla fine di quel file sorgente e non tiene conto dei blocchi di codice

nome ha la stessa sintassi di un identificatore (nome di variabile), non può contenere spazi e viene per convenzione scritto in maiuscolo

#undef nome annulla una **#define** precedente

Inclusione condizionale

Permette di include o escludere parte del codice dalla compilazione e dal preprocessing stesso

```
#if espressione_1
    istruzioni
#elif espressione_2
    istruzioni
...
#else
    istruzioni
#endif
```

Solo uno dei gruppi di `istruzioni` sarà elaborato dal preprocessore e poi compilato



Inclusione condizionale

Le **espressioni** devono essere costanti intere (non possono contenere `sizeof()`, `cast`), sono considerate vere se `!=0`

L'espressione `defined(nome)` produce 1 se `nome` è stato definito (con `#define`), 0 altrimenti

`#ifdef nome` equivale a:

`#if defined(nome)` e verifica che `nome` sia definito

`#ifndef nome` equivale a:

`#if !defined(nome)` e verifica che `nome` non sia definito

Inclusione condizionale



Nel caso in cui un file incluso ne includa a sua volta altri, per evitare di includere più volte lo stesso file, si può usare lo schema seguente (quello che segue è il file `hdr.h`):

```
#ifndef HDR
#define HDR
...
contenuto di <hdr.h>
#endif
```

Se venisse incluso una seconda volta, il simbolo `HDR` sarebbe già definito e il contenuto non verrebbe nuovamente incluso nella compilazione

Per escludere dalla compilazione un grosso blocco di codice (anche con commenti):

```
#if 0
codice da non eseguire
#endif
```

Per isolare istruzioni da usare solo per il debug:

```
#ifdef DEBUG
printf("Valore di x: %d\n", x);
#endif
```

Esercizio – Operazioni sulle frazioni



Realizzare un programma che permetta di eseguire **somme**, **sottrazioni**, **moltiplicazioni** e **divisioni** tra coppie di **frazioni** e di visualizzare il risultato in forma ridotta. Una frazione è una coppia di interi **n1** (**numeratore**), **d1** (**denominatore positivo**).

Scrivere le seguenti funzioni:

- **leggi_frazione** – Gestisce l'inserimento di una frazione da tastiera nella forma intero/intero positivo. Controlla che l'inserimento sia corretto, altrimenti chiede all'utente di reinserire i dati. Ritorna la frazione come coppia di interi.
- **leggi_operazione** – Gestisce l'inserimento da tastiera dell'operazione aritmetica da effettuare tra le due frazioni inserite. Legge il carattere inserito dall'utente, salta caratteri \n e permette il re-inserimento in caso di errore.
- **somma_frazioni** – Somma frazioni rappresentate da coppie di interi. La frazione somma è data in output attraverso i parametri della funzione.
- **moltiplica_frazioni** – Come sopra ma per il prodotto.
- **massimo_comun_divisore** – Restituisce il massimo comun divisore tra numeratore e denominatore di una frazione.
- **riduci_frazione** – Riduce una frazione dividendo il suo numeratore e denominatore per il loro massimo comun divisore.
- **stampa_frazione** – visualizza la frazione come numeratore/denominatore.
- **main** – gestisce un menu a scelta multipla per la selezione da parte dell'utente di effettuare un' operazione di somma '+', sottrazione '-', moltiplicazione '*' e divisione '/'.



Esercizi



1. Zero di una funzione v.1.0

Analizziamo ora il problema di "risolvere", in campo reale, un'equazione ad una incognita, cioè di trovare il passaggio per lo zero di una funzione qualunque.

Teorema dello zero

Una funzione continua che assume valori di segno opposto agli estremi di un intervallo ammette almeno uno zero nell'intervallo.

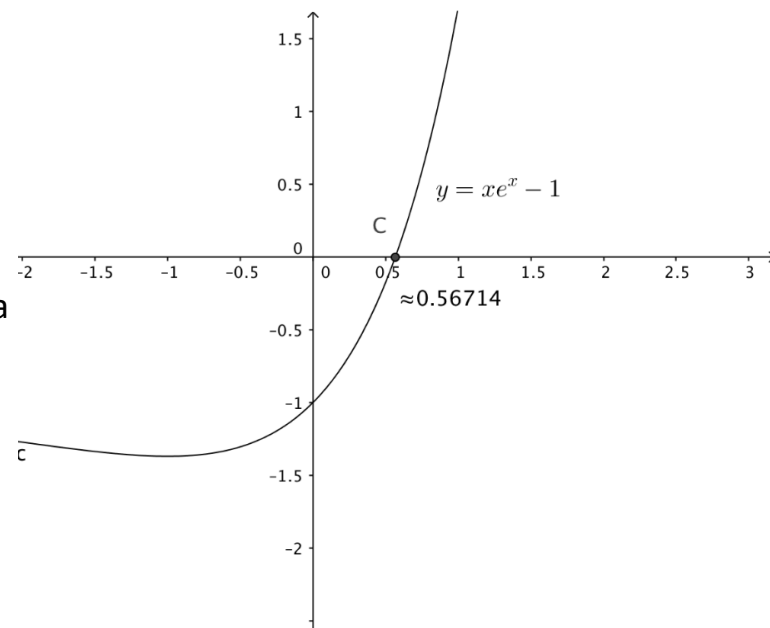
Definizione dell'intervallo

Due sono i criteri utilizzati normalmente:

- 1) Espansione di un intervallo "piccolo" fino a trovare che la funzione cambia di segno.
- 2) Partizionamento di un intervallo "grande" in intervalli più piccoli, fino a trovare uno (o più) intervalli agli estremi dei quali la funzione cambia di segno.

Tolleranza

La precisione richiesta nella determinazione dello zero stesso. Gli algoritmi di ricerca degli zeri si aspettano di ricevere dall'utente l'errore di misura tollerato.



Esercizi - Zero di una funzione v.1.0



Il metodo di bisezione

Il criterio di ricerca degli zeri più semplice consiste nel continuare a dividere in due l'intervallo di ricerca iniziato, e continuare a scegliere l'intervallo agli estremi del quale la funzione cambia segno.



Zero di una funzione v.1.0

Non è un criterio molto efficiente, ma sicuramente converge, anche se la funzione è discontinua o ha punti di singolarità. Se nell'intervallo ci sono più zeri, convergerà ad uno di questi.

