



Le funzioni

Prof. Marco Gavanelli

Dipartimento di Ingegneria

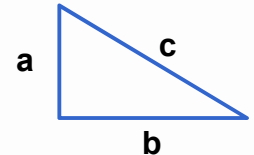
Università di Ferrara

1

Funzioni di libreria

- Il linguaggio C ha il concetto di funzione. Molte funzioni utili sono già predefinite.
- Ad esempio, includendo `#include<math.h>` si possono usare `sin` (seno), `cos` (coseno), `sqrt` (radice quadrata), ecc.

```
#include <stdio.h>
#include <math.h>
main()
{ float a, b, c;
  scanf("%f %f",&a, &b);
  c = sqrt(a*a+b*b);
  printf("la diagonale e` %f", c);
```



2



Funzioni di libreria

- Le funzioni si usano all'interno delle espressioni, come gli operatori del linguaggio C (come +, -, *, ...)

`c = sqrt(a*a+b*b);`

- invocazione della funzione:

- *nome della funzione*: `sqrt`
- *parametri attuali*: possono essere:
 - costanti: `x = sqrt(2);`
 - variabili: `x = sqrt(y);`
 - espressioni: `x = sqrt(3*y+1);`

3

Esempio

- Dati due numeri *a* e *b*, si calcoli a^b e b^a

```
main()
{ int a, b, i, p1;
  scanf("%d %d",&a,&b);

  p1=1;
  for (i=0;i<b;i++)
    p1=p1*a;
  p1=1;
  for (i=0;i<a;i++)
    p1=p1*b;
```

} copia

} incolla

4



Esempio

- Dati due numeri a e b , si calcoli a^b e b^a

```
main()
{ int a, b, i, p1, p2;
  scanf("%d %d",&a,&b);
  p1=1;
  for (i=0;i<b;i++)
    p1=p1*a;
  p2=1;
  for (i=0;i<a;i++)
    p2=p2*b;
  printf("%d %d",p1,p2);}
```

Modifica a
mano delle
differenze

5

Meglio con le funzioni

```
int power(int a, int b)
{ int i,p=1;
  for (i=0;i<b;i++)
    p = p*a;
  return p;
}
```

definizione

```
main()
{ int a,b;
  scanf("%d %d",&a,&b);
  printf("%d %d",power(a,b),power(b,a));
}
```

invocazioni

6

SOTTOPROGRAMMI



- Un sottoprogramma è una nuova istruzione, o un nuovo operatore definito dal programmatore per sintetizzare una sequenza di istruzioni.
- In particolare:
 - procedura**: è un sottoprogramma che rappresenta un'istruzione non primitiva
 - funzione**: è un sottoprogramma che rappresenta un operatore non primitivo.
- Tutti i linguaggi di alto livello offrono la possibilità di definire funzioni e/o procedure.

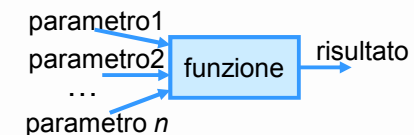
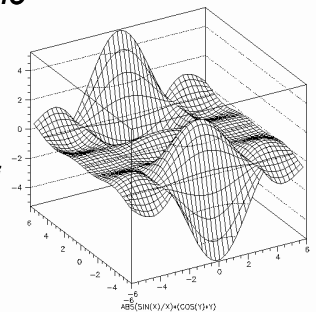
Il linguaggio C realizza solo il concetto di **funzione**.

7

FUNZIONI COME COMPONENTI SW



- Una **funzione** è un componente software che cattura l'idea matematica di funzione
 - molti possibili ingressi (che non vengono modificati!)
 - una sola uscita (il risultato)
- Una funzione
 - riceve dati di ingresso in corrispondenza ai parametri
 - ha come corpo una espressione, la cui valutazione fornisce un risultato
 - denota un valore in corrispondenza al suo nome



8

DEFINIZIONE DI FUNZIONE



```
<definizione-di-funzione> ::=  
<tipoValore> <nome> (<parametri-formali>)  
{  
    <corpo>  
}  
  
<parametri-formali>  
    – o una lista vuota: void  
    – o una lista di variabili (separate da virgole) visibili solo  
      entro il corpo della funzione.  
  
<tipoValore>  
    – deve coincidere con il tipo del valore risultato della funzione
```

9

DEFINIZIONE DI FUNZIONE



```
<definizione-di-funzione> ::=  
<tipoValore> <nome> (<parametri-formali>)  
{  
    <corpo>  
}
```

- Nella parte **corpo** possono essere presenti definizioni e/o dichiarazioni locali (parte **dichiarazioni**) e un insieme di istruzioni (parte **istruzioni**).
- I dati riferiti nel corpo possono essere **costanti**, **variabili**, oppure **parametri formali**.
- All'interno del corpo, i parametri formali vengono trattati come variabili.

10

RISULTATO DI UNA FUNZIONE



- L'istruzione **return** serve a dichiarare qual è il risultato, il valore denotato dalla funzione

return <espressione>;

il valore dell'espressione viene fornito come risultato e chiamato valore di ritorno della funzione

- L'esecuzione dell'istruzione **return** provoca anche la terminazione dell'esecuzione della funzione



- Eventuali istruzioni successive alla **return** non saranno mai eseguite!

11

ESEMPIO



```
int power(int a, int b) } interfaccia  
{ int i, p=1;  
  for (i=0; i<b; i++)  
    p = p*a;  
  return p;  
}
```

- Il simbolo **power** denota il nome della funzione
- Le variabili intere **a** e **b** sono i parametri (formali) della funzione
- Il valore restituito è un intero **int**.

12

CHIAMATA DI FUNZIONE



- *La chiamata (o invocazione) di funzione è un'espressione della forma*

`<nomefunzione> (<parametri-attuali>)`

dove:

```
<parametri-attuali> ::=  
[ <espressione> ] { , <espressione> }
```

- *I parametri attuali devono corrispondere ai parametri formali*
 - Come numero
 - Come tipo

13

Invocazione delle funzioni



- *Una volta definita una funzione, la posso invocare più volte, anche all'interno delle espressioni*

A tutti gli effetti, **power** è un nuovo operatore che posso usare nel programma!

```
main()  
{ int a,b,c,Delta;  
  scanf("%d %d %d",&a,&b,&c);  
  Delta = power(b,2) - 4*a*c;  
  printf("%d",Delta);  
}
```

14

Esercizio



- *Si leggano da tastiera due numeri interi a e b, controllando che siano entrambi compresi fra 0 e 10.*
- *Se l'utente inserisce un valore esterno all'intervallo 0-10 si mostri un messaggio di errore e si faccia inserire nuovamente il numero*
- *Si stampi poi il maggiore dei due*

15

Vantaggi



- *Scrivo il codice una volta sola*
- *Il codice è più chiaro:*
 - *Se do nomi significativi alle funzioni, spiegano già che cosa fanno, ho meno bisogno di commenti, spiegazioni, ...*

16



```
int power(int a, int b)
{
    int i, p=1;
    for (i=0; i<b; i++)
        p = p*a;
    return p;
}

main()
{
    int x=3, y=2, i;
    i = power(x, y);
}
```

• Il **main** (e le altre funzioni) non possono modificare il valore delle variabili locali

Queste due variabili si chiamano entrambe **i**, ma sono variabili diverse

• questo permette di scrivere il **main** disinteressandosi da come è realizzata la funzione **power**

17



- Poiché ogni funzione può modificare solo le sue variabili, posso modificare ogni funzione indipendentemente
 - correggere errori
 - aggiungere funzionalità
 - usare algoritmi più efficienti
- Devo solo stare attento a non modificare l'interfaccia
- L'interfaccia contiene tutte le informazioni che servono a chi vuole invocare la funzione

18

Esercizio



- Dati due numeri a e b , si calcoli a^b e b^a
- Si modifichi ora il programma in modo che funzioni anche con esponenti negativi

<pre>float power(int a, int b) { int i; float p=1; if (b>=0) for (i=0; i<b; i++) p = p*a; else for (i=0; i< -b; i++) p = p/a; return p; }</pre>	<pre>main() { int a, b; scanf("%d %d", &a, &b); printf("%f ", power(a, b)); printf("%f ", power(b, a)); }</pre>
--	---

19

Vantaggi



- Se devo effettuare una **modifica**, la faccio in un solo punto
 - Es: se voglio considerare anche potenze negative?
 - se trovo un algoritmo più efficiente?
 - se trovo un errore nell'algoritmo?
- Non devo modificare il **main**, a meno che non cambi l'interfaccia della funzione

20



- **Riutilizzo del codice**
 - Se voglio utilizzare la funzione potenza, posso fare un copia-incolla in un altro programma (senza dovermi preoccupare di cambiare i nomi delle variabili, ...)
 - Posso dare il mio codice ad altri, o usare il codice fatto da altri, senza dover guardare l'algoritmo, controllare che non ci siano conflitti con i nomi delle variabili, ...
 - Il codice che ho scritto e testato in un'altra applicazione probabilmente è corretto
- **Suddivisione del lavoro: varie persone possono collaborare indipendentemente, mettendosi d'accordo sulle interfacce**
- **Utilizzo meno memoria (il sorgente è più corto, ed anche il compilato)**

21

Esempio

```
int power(int a, int b)
{ int i,p=1;
  for (i=0;i<b;i++)
    p = p*a;
  return p;
}

main()
{ int x=3,y=2,k,j;
  k = power(x,y);
  j = power(y,x+1);
}
```

23

Esercizio

- **Si scriva una funzione che verifica se un anno è bisestile**
- **Si legga poi da tastiera una data e si dica se è valida**

22

Modello Cliente-Servitore

- **Il meccanismo di uso di funzioni nei linguaggi di programmazione fa riferimento allo schema di interazione tra componenti software**

cliente – servitore
(client – server)

- **Es: segreteria studenti**
 - Lo studente fornisce alla persona allo sportello:
 - domanda di iscrizione
 - fototessera
 - la carta d'identità
 - il certificato di maturità
 - bollettino pagamento tasse
 - La persona allo sportello fornisce
 - libretto
 - nome utente/password per la posta elettronica
 - Lo studente **si disinteressa completamente** di che cosa viene effettuato dalla segreteria; gli interessa solo sapere come accedere al servizio:
 - qual è lo sportello giusto
 - quali informazioni deve fornire alla segreteria
 - quali informazioni gli vengono date in cambio

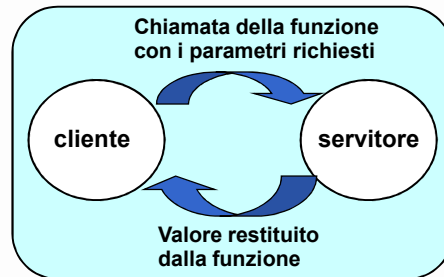


24

INTERFACCIA DI UNA FUNZIONE

- L'interfaccia (o firma o signature) di una funzione comprende

- nome della funzione
- lista dei parametri
- tipo del valore da essa denotato

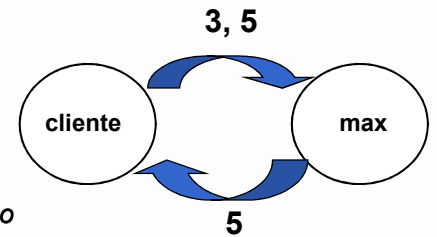


- Esplicita il contratto di servizio fra cliente e servitore.
- Cliente e servitore comunicano quindi mediante
 - i **parametri** trasmessi dal cliente al servitore all'atto della chiamata (direzione: dal cliente al servitore)
 - il **valore restituito** dal servitore al cliente direzione: dal servitore al cliente)

25

ESEMPIO

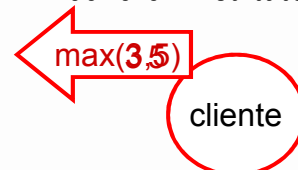
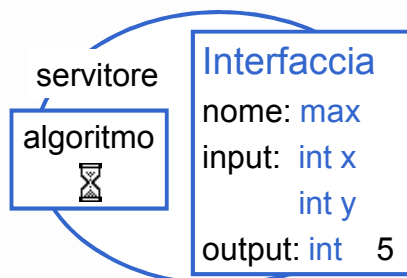
- **Calcolo del massimo di due valori**
- **Sequenza operazioni:**
 - Il cliente comunica al servitore i due valori
 - Il servitore calcola il massimo
 - Il servitore comunica al cliente il valore del massimo
- Cliente e servitore devono essere d'accordo su:
 - il tipo dei valori → int
 - quanti sono i valori → 2
 - qual è il nome del servitore → max
 - qual è il tipo del valore di ritorno → int
- Il cliente non è interessato all'algoritmo che il servitore utilizza



26

ESEMPIO

- Quindi, dal lato del servitore dovrà esserci scritto:
 - come si chiama
 - di quanti dati ha bisogno e di che tipo
 - di che tipo è il valore calcolato
- Il cliente dovrà mandare al servitore identificato da quel nome
 - i dati di ingresso, in numero e tipo giusti
 - ricevere il risultato

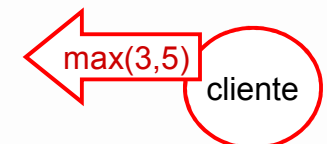
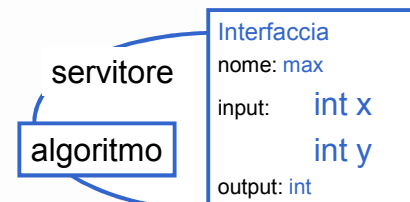


27

COMUNICAZIONE CLIENTE → SERVITORE

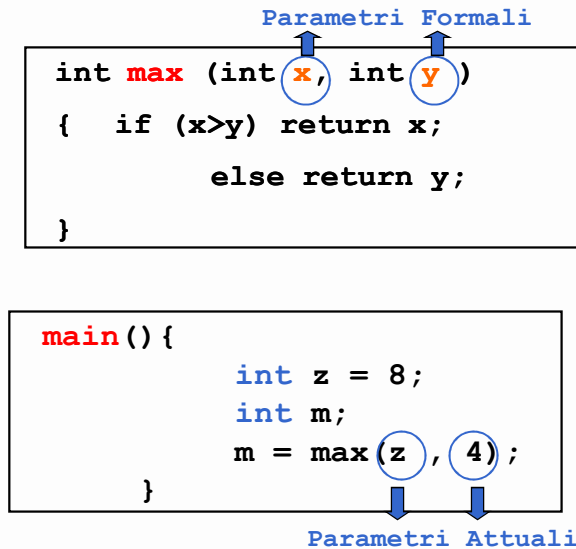
Il cliente passa informazioni al servitore tramite una serie di **Parametri**

- **Parametri formali :**
 - sono specificati nella dichiarazione del servitore
 - esplicitano il contratto fra servitore e cliente
 - indicano cosa il servitore si aspetta dal cliente
- **Parametri attuali :**
 - sono trasmessi dal cliente all'atto della chiamata
 - devono corrispondere ai parametri formali in numero, posizione e tipo



28

ESEMPIO



29

COMUNICAZIONE CLIENTE/SERVITORE

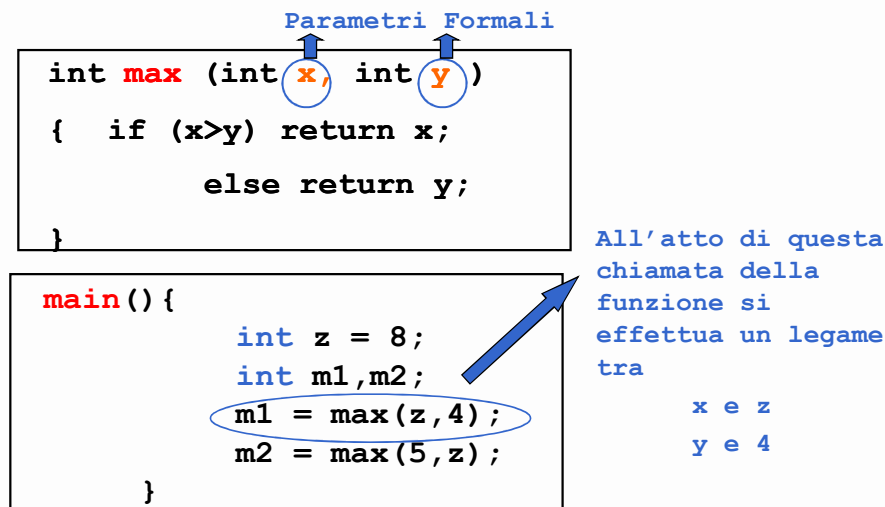
- *Legame tra parametri attuali e parametri formali: effettuato al momento della chiamata, in modo dinamico.*

Tale legame:

- *vale solo per l'invocazione corrente*
- *vale solo per la durata della funzione.*

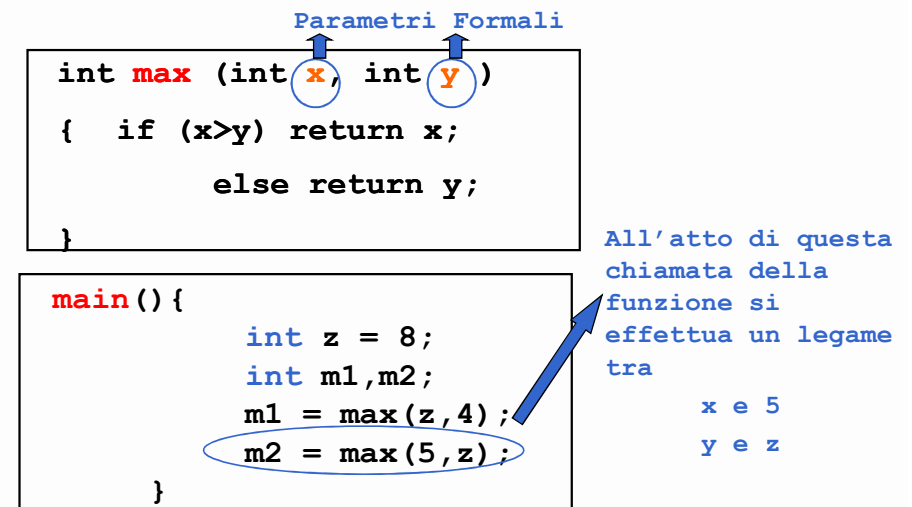
30

ESEMPIO



31

ESEMPIO



32

INFORMATION HIDING

- La struttura interna (corpo) di una funzione è completamente inaccessibile dall'esterno.
- Così facendo si garantisce protezione dell'informazione (information hiding)
- Una funzione è accessibile SOLO attraverso la sua interfaccia.
- Quindi posso cambiare l'algoritmo della funzione senza preoccuparmi di quello che succede nel programma che la invoca

33

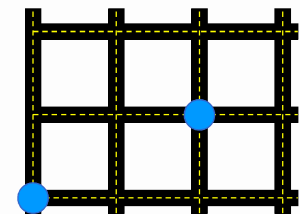
Esercizio

- Scrivere una funzione che prende in ingresso due strutture di tipo punto

```
typedef struct
```

```
{ int x,y; } punto;
```

e calcola la distanza di Manhattan fra i due punti



34

Esercizio (adattato dal compito 9 gen 07)

- Sia data la definizione

```
typedef struct {int x,y; } punto;
```
- Si definisca una struttura che rappresenta una figura geometrica. La struttura contiene:
 - **Tipo**: stringa che può essere "CERCHIO" o "QUADRATO"
 - **P**: punto che rappresenta
 - il centro, nel caso del cerchio
 - il punto in basso a sinistra, nel caso del quadrato
 - **L**: lunghezza
 - del raggio, nel caso del cerchio
 - del lato, nel caso del quadrato
- Si scriva una funzione che prende in ingresso una struttura **figura** e un **punto** e fornisce
 - 1 se il punto è interno alla figura
 - 0 se il punto è esterno



35

Modello a run-time delle funzioni

- Per eseguire una funzione bisogna:
 1. creare le variabili
 2. ricopiare il valore dei parametri
 3. eseguire il codice della funzione
 4. restituire il risultato
 5. liberare l'area di memoria che conteneva le variabili

36

Tempo di vita

- Il tempo di vita (lifetime) di una variabile è l'intervallo di tempo in cui esiste un'area di memoria associata alla variabile
- In particolare, le variabili che sono definite all'interno di una funzione hanno, come **tempo di vita**, quello in cui esiste la funzione, da quando viene invocata a quando esegue l'istruzione `return`.

37

Creazione delle variabili

- Le variabili necessarie all'esecuzione della funzione sono
 - i parametri
 - le variabili locali
- Queste vengono inserite in una struttura dati detta **record di attivazione** della funzione

38

FUNZIONI COME COMPONENTI SW: NASCITA E MORTE

- All'atto della chiamata, **l'esecuzione del cliente viene sospesa e il controllo passa al servitore**.
- Il servitore "vive" solo per il tempo necessario a svolgere il servizio.
- Al termine, il servitore "muore", e l'esecuzione torna al cliente.
- In particolare, le variabili che erano definite all'interno della funzione hanno, come **tempo di vita**, quello in cui esiste la funzione, da quando viene invocata a quando esegue l'istruzione `return`.

39

IL MODELLO A RUN-TIME

- Nel momento in cui una funzione viene invocata, è necessario **creare** delle nuove variabili
- Bisognerà richiedere dello spazio in memoria per allocare
 - i parametri formali
 - le variabili locali
- Viene creato il **Record di Attivazione** della funzione

40

RIASSUMENDO...

All'atto dell'invocazione di una funzione:

- si crea una nuova attivazione (istanza) del servitore
- si alloca la memoria per i parametri (e le eventuali variabili locali)
- si trasferiscono i parametri al servitore
- si trasferisce il controllo al servitore
- si esegue il codice della funzione.

41

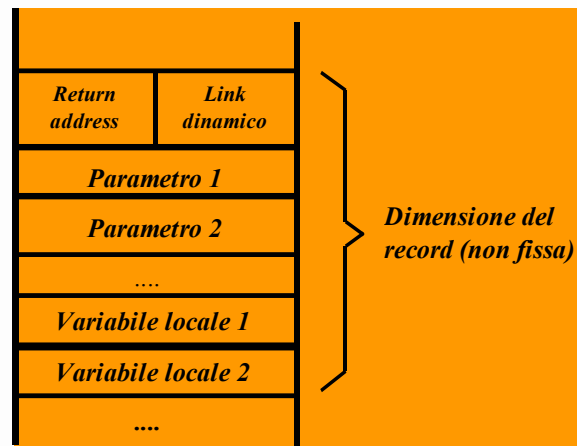
RECORD DI ATTIVAZIONE

È il “mondo della funzione”: contiene tutto ciò che ne caratterizza l'esistenza

- i **parametri ricevuti**
- le **variabili locali**
- l'**indirizzo di ritorno (Return address RA)** che indica il punto a cui tornare al termine della funzione, per permettere al programma di proseguire una volta che la funzione termina. È l'indirizzo dell'istruzione in linguaggio macchina da eseguire subito dopo l'esecuzione della funzione
- un **collegamento al record di attivazione della funzione invocante (Link Dinamico DL)**

42

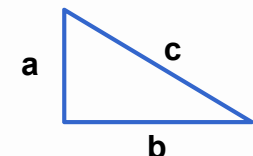
RECORD DI ATTIVAZIONE



43

Esercizio

- Si scriva un programma C che legge da tastiera 3 numeri e verifica se questi costituiscono una terna pitagorica.
- Si supponga che l'utente inserisca i numeri in ordine dal più piccolo al più grande



44

Esercizio

- Si scriva una funzione `min_abs` che, dati due interi, fornisce il minimo dei loro valori assoluti
- Esempio:

```
main()  
{  int a=7, b=-2, m;  
    m = min_abs(a,b+1);  
}
```


alla fine del programma, `m` vale 1.

45

RECORD DI ATTIVAZIONE

- Rappresenta il “mondo della funzione”: nasce e muore con essa
 - è creato al momento della invocazione di una funzione
 - permane per tutto il tempo in cui la funzione è in esecuzione
 - è distrutto (deallocato) al termine dell'esecuzione della funzione stessa.
- Ad ogni chiamata di funzione viene creato un nuovo record, specifico per quella chiamata di quella funzione
- La dimensione del record di attivazione
 - varia da una funzione all'altra
 - per una data funzione, è fissa e calcolabile a priori

46

RECORD DI ATTIVAZIONE

- Funzioni che chiamano altre funzioni *danno luogo a una sequenza di record di attivazione*
 - allocati secondo l'ordine delle chiamate
 - deallocati in ordine inverso
- La sequenza dei link dinamici costituisce la cosiddetta catena dinamica, che rappresenta la storia delle attivazioni (“chi ha chiamato chi”)

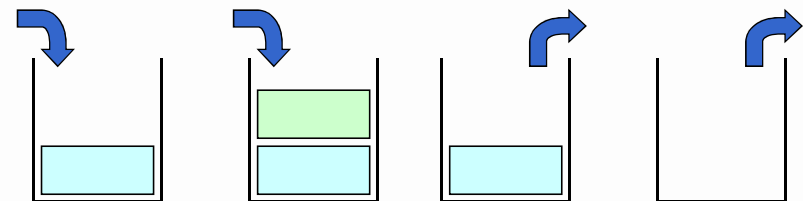
47

RECORD DI ATTIVAZIONE

- Per catturare la semantica delle chiamate annidate (una funzione che chiama un'altra funzione che...), l'area di memoria in cui vengono allocati i record di attivazione deve essere gestita come una pila

STACK

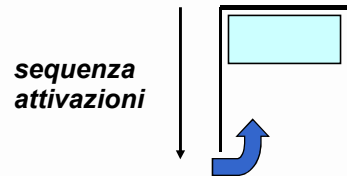
Una struttura dati gestita con politica LIFO (Last In, First Out - l'ultimo a entrare è il primo a uscire)



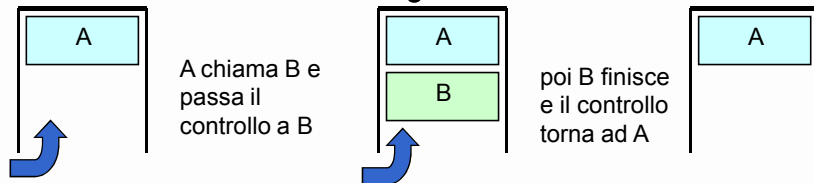
48

RECORD DI ATTIVAZIONE

- Normalmente lo STACK dei record di attivazione si disegna nel modo seguente



- Quindi, se la funzione A chiama la funzione B lo stack evolve nel modo seguente



49

RECORD DI ATTIVAZIONE

Il valore di ritorno calcolato dalla funzione può essere restituito al cliente *in due modi*:

- **inserendo un apposito “slot” nel record di attivazione**
 - il cliente deve copiarci il risultato da qualche parte prima che il record venga distrutto
- **tramite un registro della CPU**
 - soluzione più semplice ed efficiente, privilegiata ovunque possibile.

50

ESEMPIO DI CHIAMATE ANNIDATE

Programma:

```
int R(int A) { return A+1; }
int Q(int x) { return R(x); }
int P(void) { int a=10; return Q(a); }
main() { int x = P(); }
```

Sequenza chiamate:

S.O. → main → P() → Q() → R()

51

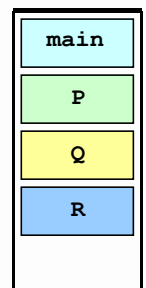
ESEMPIO DI CHIAMATE ANNIDATE

```
int R(int A)
{ return A+1; }

int Q(int x)
{ return R(x); }

int P(void)
{ int a=10; return Q(a); }

main()
{ int x = P(); }
```



52

Funzioni e strutture

- Con le funzioni si possono usare come parametri e come valore di ritorno le strutture (con gli array è un po' diverso, come vedremo)
- L'utilizzo delle funzioni permette di costruire semplicemente applicazioni con la metodologia top-down
- Si scriva un programma che permette di
 - leggere due frazioni
 - calcolarne la somma
 - visualizzare il risultato

$$\frac{1}{6} + \frac{1}{3} = \frac{1}{2}$$

53

Esempio

- Per prima cosa, definiamo le strutture dati:
 - una frazione è costituita da un numeratore ed un denominatore
- Poi scriviamo l'algoritmo partendo dalla versione più astratta. Scriviamo il main invocando le varie funzioni che ci servono

```
typedef struct { int num; int den; } frazione;

main()
{   frazione f1, f2, somma;
    f1 = leggiFrazione();
    f2 = leggiFrazione();
    somma = sum(f1,f2);
    printf("%d/%d", somma.num, somma.den);
}
```

54

Esempio

- Poi implementiamo le funzioni che abbiamo invocato nel main:

```
frazione leggiFrazione()
{   frazione f;
    scanf("%d",&f.num);
    scanf("%d",&f.den);
    return f;
}
```

55

Esempio

- La somma di due frazioni si calcola così:
 - calcolo il minimo comun denominatore (minimo comune multiplo dei denominatori); questo è il denominatore della somma
 - porto la prima frazione al comun denominatore
 - porto la seconda frazione al comun denominatore
 - calcolo la somma dei numeratori: questo è il numeratore della somma

```
frazione sum(frazione f1, frazione f2)
{   int mcd; // minimo comun denominatore
    frazione somma;
    mcd = mcm(f1.den, f2.den);
    somma.den = mcd;
    f1 = portaDen(f1, mcd);
    f2 = portaDen(f2, mcd);
    somma.num = f1.num + f2.num;
    return somma;
}
```

56

Esempio

- Infine implementiamo le funzioni che abbiamo usato nelle funzioni
- Per portare una frazione ad un denominatore, devo moltiplicare numeratore e denominatore per la stessa quantità

$$\text{nuovoNum}/\text{nuovoDen} = \text{vecchioNum}/\text{vecchioDen}$$

- quindi

$$\text{nuovoNum} = \text{vecchioNum} * \text{nuovoDen} / \text{vecchioDen}$$

```
frazione portaDen(frazione f, int nDen)
{
    frazione nuovo;
    nuovo.den = nDen;
    nuovo.num = f.num*nDen/f.den;
    return nuovo;
}
```

57

Esempio

- Per calcolare il minimo comune multiplo di due interi, posso farne il prodotto e dividere per il massimo comun divisore dei due

```
int mcm(int a, int b)
{
    return a*b/MCD(a,b);
}
```

58

Esempio

- Per calcolare il MCD di due numeri, posso usare il metodo di Euclide

```
int MCD(int m, int n)
{
    while (m != n)
        if (m>n)
            m=m-n;
        else n=n-m;
    return m;
}
```



59

Il programma risultante

```
#include <stdio.h>
typedef struct { int num; int den; } frazione;

int MCD(int m, int n)
{
    while (m != n)
        if (m>n)
            m=m-n;
        else n=n-m;
    return m;
}

int mcm(int a, int b)
{
    return a*b/MCD(a,b);
}

frazione portaDen(frazione f, int nDen)
{
    frazione nuovo;
    nuovo.den = nDen;
    nuovo.num = f.num*nDen/f.den;
    return nuovo;
}

frazione sum(frazione f1, frazione f2)
{
    int mcd;
    frazione somma;
    mcd = mcm(f1.den, f2.den);
    somma.den = mcd;
    f1 = portaDen(f1, mcd);
    f2 = portaDen(f2, mcd);
    somma.num = f1.num + f2.num;
    return somma;
}

frazione leggiFrazione()
{
    frazione f;
    scanf("%d", &f.num);
    scanf("%d", &f.den);
    return f;
}

main()
{
    frazione f1, f2, somma;
    f1 = leggiFrazione();
    f2 = leggiFrazione();
    somma = sum(f1, f2);
    printf("%d/%d", somma.num, somma.den);
}
```

Il programma risultante

- E' abbastanza facile da scrivere e da capire
- E' facile da modificare
- Es: voglio assicurarmi che l'utente non inserisca una frazione che ha per denominatore zero
- Intervengo in una sola funzione: la `leggiFrazione`
 - è una funzione di 4 istruzioni, quindi facile da capire e da modificare

```
frazione leggiFrazione()
{
    frazione f;
    do
    {
        scanf("%d",&f.num);
        scanf("%d",&f.den);
        if (f.den==0)
            printf("Re-inserire la frazione\n");
    } while (f.den == 0);
    return f;
}
```

61

Modificabilità

- Ora voglio che mi fornisca solo frazioni ai minimi termini
- Aggiungo una funzione `riduci`. Posso invocarla nel `main`

```
main()
{
    frazione f1, f2, somma;
    f1 = leggiFrazione();
    f2 = leggiFrazione();
    somma = riduci(sum(f1,f2));
    printf("%d/%d",somma.num,somma.den);
}
```

62

Modificabilità

- Poi definisco la nuova funzione `riduci`
- Per ridurre una frazione ai minimi termini, basta dividere numeratore e denominatore per il loro MCD

```
frazione riduci(frazione f)
{
    int m = MCD(f.num,f.den);
    f.num = f.num/m;
    f.den = f.den/m;
    return f;
}
```

63

PASSAGGIO DEI PARAMETRI IN C

In C, i parametri sono trasferiti sempre e solo per valore (by value o per copia)

- si trasferisce una copia del parametro attuale, non l'originale!
- tale copia è strettamente privata e locale a quel servitore
- il servitore potrebbe quindi alterare il valore ricevuto, senza che ciò abbia alcun impatto sul cliente

64

PASSAGGIO DEI PARAMETRI IN C

*In C, i parametri sono trasferiti sempre e solo **per valore** (by value o **per copia**)*

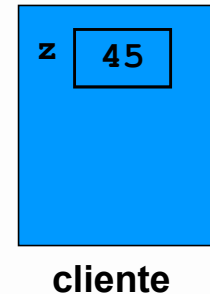
Conseguenza:

- *è impossibile usare un parametro per trasferire informazioni verso il cliente*
- *per trasferire un'informazione al cliente si sfrutta il valore di ritorno della funzione*

65

PASSAGGIO PER VALORE

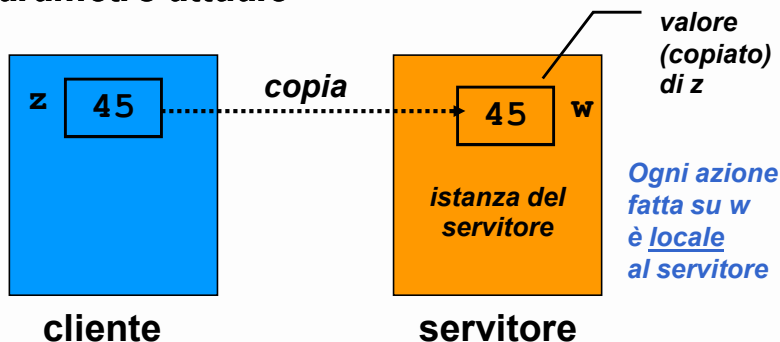
- *si trasferisce una copia del valore del parametro attuale*



66

PASSAGGIO PER VALORE

- *si trasferisce una copia del valore del parametro attuale*



67

ESEMPIO: VALORE ASSOLUTO

- **Definizione formale:**

$$\begin{aligned} |x| : \mathbb{Z} &\rightarrow \mathbb{N} \\ |x| &\text{ vale } x && \text{se } x \geq 0 \\ |x| &\text{ vale } -x && \text{se } x < 0 \end{aligned}$$

- **Codifica sotto forma di funzione C:**

```
int valAss(int x)
{
    if (x<0) return -x;
    else return x;
}
```

68

```

int valAss(int x)
{ if (x<0) x = -x;
  return x;
}

main()
{ int absz, z = -87;
  absz = valAss(z);
  printf("|%d|=%d", z, absz);
}

```

69

ESEMPIO: VALORE ASSOLUTO

• Servitore

```

int valAss(int x)
{ if (x<0) x = -x;
  return x;
}

```

x -87

• Cliente

```

main()
{ int absz, z = -87;
  absz = valAss(z);
  printf("|%d|=%d", z, absz);
}

```

Quando valAss(z) viene chiamata, il valore attuale di z, valutato nell'environment corrente (-87), viene copiato e passato a valAss. Quindi x vale -87

70

ESEMPIO: VALORE ASSOLUTO

• Servitore

```

int valAss(int x)
{ if (x<0) x = -x;
  return x;
}

```

x ~~-87~~ 87

valAss restituisce il valore 87 che viene assegnato a absz

• Cliente

```

main()
{ int absz, z = -87;
  absz = valAss(z);
  printf("|%d|=%d", z, absz);
}

```

NOTA: IL VALORE DI z NON VIENE MODIFICATO

71

ESEMPIO: VALORE ASSOLUTO

• Servitore

```

int valAss(int x)
{ if (x<0) x = -x;
  return x;
}

```

NOTA: IL VALORE DI z NON VIENE MODIFICATO

• Cliente

```

main()
{ int absz, z = -87;
  absz = valAss(z);
  printf("|%d|=%d", z, absz);
}

```

la printf stampa |-87| = 87

72

PASSAGGIO DEI PARAMETRI IN C

Vantaggi:

- Evita di effettuare modifiche “per sbaglio” sulle variabili del cliente

Limiti:

- consente di restituire al cliente solo valori di tipo (relativamente) semplice
- non consente di restituire collezioni di valori
- non consente di scrivere componenti software il cui scopo sia diverso dal calcolo di una espressione

73

PASSAGGIO DEI PARAMETRI IN C

Il C non supporta direttamente il passaggio per riferimento

- è una grave mancanza!
- il C lo fornisce indirettamente solo per alcuni tipi di dati
- quindi, occorre costruirselo quando serve. (vedremo più avanti dei casi)

75

PASSAGGIO DEI PARAMETRI

Molti linguaggi mettono a disposizione il passaggio per riferimento (by reference)

- non si trasferisce una copia del valore del parametro attuale
- si trasferisce un riferimento al parametro, in modo da dare al servitore accesso diretto al parametro in possesso del cliente
 - il servitore accede e modifica direttamente il dato del cliente.

74

Esercizio

- Calcolare la funzione seno di x con la seguente formula (fermarsi ad un esponente dato come parametro):

$$\text{sen } x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}.$$

- Il fattoriale di un numero è definito come
 - $n! = 1$ se $n = 0$
 - $n! = n(n-1)!$ se $n > 0$
- In pratica, si può calcolare come
 - $n! = 1 \times 2 \times 3 \times \dots \times n$

76

Soluzione

- Ragionamento bottom-up:
- Possiamo riutilizzare le funzioni che già conosciamo: la potenza di un numero
- Ci manca la funzione fattoriale

77

Fattoriale (iterativo)

```
int fact(int n)
{ int i,f=1;
  for (i=1;i<=n;i++)
    f=f*i;
  return f;
}
```

78

Ora possiamo definire la funzione sen

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}.$$

```
float sen(float x, int precisione)
{ int i, segno=1;
  float ris=0;
  for (i=1;i<precisione;i=i+2)
  {
    ris=ris+segno*power(x,i)/fact(i);
    segno = -segno;
  }
  return ris;
}
```

79

Ordine delle funzioni

- Per il C, vale una regola fondamentale:

“Un identificatore non è visibile prima della sua dichiarazione”

- L’abbiamo già visto per le variabili: lo stesso concetto vale anche per le funzioni

80

Quindi ...

```
int power(int a, int b)
{
    int i, p=1;
    for (i=0; i<b; i++)
        p = p*a;
    return p;
}

int fact(int n)
{
    int i, f=1;
    for (i=1; i<n; i++)
        f=f*i;
    return f;
}

float sen(float x, int precisione)
{
    int i, segno=1;
    float ris=0;
    for (i=1; i<precisione; i=i+2)
    {
        ris=ris+segno*power(x,i)/fact(i);
        segno = -segno;
    }
    return ris;
}

main()
{
    printf("%f", sen(0.1,4));
}
```

Ok!

```
main()
{
    printf("%f", sen(0.1,4));
}

int power(int a, int b)
{
    int i, p=1;
    for (i=0; i<b; i++)
        p = p*a;
    return p;
}

float sen(float x, int precisione)
{
    int i, segno=1;
    float ris=0;
    for (i=1; i<precisione; i=i+2)
    {
        ris=ris+segno*power(x,i)/fact(i);
        segno = -segno;
    }
    return ris;
}

int fact(int n)
{
    int i, f=1;
    for (i=1; i<n; i++)
        f=f*i;
    return f;
}
```

No!

81

Prototipi

- In realtà, per riuscire ad invocare il servitore, il cliente ha bisogno solo dell'interfaccia
- Per questo, si può definire la sola interfaccia e poi dare la definizione della funzione più avanti.

82

DICHIARAZIONE DI FUNZIONE

La **dichiarazione** di una funzione (o **prototipo della funzione**) è costituita dalla **sola interfaccia**, senza corpo (sostituito da un **;**)

<dichiarazione-di-funzione> ::=
<tipoValore> <nome> (<parametri>);

83

Esempio

```
int power(int a, int b);

main()
{
    printf("%d", power(2,3));
}
```

Ok!

```
int power(int a, int b)
{
    int i, p=1;
    for (i=0; i<b; i++)
        p = p*a;
    return p;
}
```

84

DICHIARAZIONE DI FUNZIONI

- La **definizione** di una funzione costituisce l'**effettiva realizzazione** del componente
 - Dice come è fatto il componente
- La **dichiarazione** specifica il **contratto di servizio** fra la funzione e chi la deve invocare, esprimendo le proprietà essenziali della funzione.
 - Dice come si usa il componente
 - Per usare una funzione non è necessario sapere come è fatta, anzi, è controproducente

85

Esempio

```
int power(int a, int b);
```

} **dichiarazione**

```
main()  
{ printf("%d",power(2,3));  
}
```

```
int power(int a, int b)  
{ int i,p=1;  
  for (i=0;i<b;i++)  
    p = p*a;  
  return p;  
}
```

} **definizione**

86

DICHIARAZIONE vs. DEFINIZIONE

- La **definizione** di una funzione costituisce l'**effettiva realizzazione** del componente
 - Non può essere duplicata
 - Ogni applicazione deve contenere una e una sola definizione per ogni funzione utilizzata
 - La compilazione della definizione genera il codice macchina che verrà eseguito ogni volta che la funzione verrà chiamata.

```
int power(int a, int b)  
{ int i,p=1;  
  for (i=0;i<b;i++)  
    p = p*a;  
  return p;  
}
```

} **definizione**

87

DICHIARAZIONE vs. DEFINIZIONE

- La **dichiarazione** di una funzione costituisce **solo una specifica** delle proprietà del componente:
 - Può essere duplicata senza danni
 - Un'applicazione può contenerne più di una
 - La compilazione di una dichiarazione non genera codice macchina

```
int power(int a, int b);
```

} **dichiarazione**

88

DICHIARAZIONE vs. DEFINIZIONE

- La definizione è molto più di una dichiarazione

definizione = dichiarazione + corpo



La definizione funge anche da dichiarazione
(ma non viceversa)

89

Esempio

```
int primo(int n)
{ int i=2,divisibile=0;
  while ((!divisibile) && (i<n))
  {   divisibile = (n % i)==0;
      i++;
  }
  return (!divisibile);
}

main()
{ int n;
  scanf("%d",&n);
  if (primo(n))
      printf("%d e` primo",n);
  else printf("%d non e` primo",n);
}
```

Posso renderlo
più efficiente!

90

Esempio

```
int primo(int n)
{ int i=3,divisibile=0;
  if (n<=2) return 1;
  if (n % 2 ==0) return 0;
  else
  {   while ((!divisibile) && (i<n/2))
      {   divisibile = (n % i)==0;
          i=i+2;
      }
      return (!divisibile);
  }
}

main()
{ int n;
  scanf("%d",&n);
  if (primo(n))
      printf("%d e` primo",n);
  else printf("%d non e` primo",n);
}
```

91

Esercizio

- Si scriva una funzione in linguaggio C che calcola il valore della funzione matematica $\Pi(n)$, definita come “il numero di numeri primi compresi fra 1 ed n”

$\Pi(1)=0$

$\Pi(2)=1$ {2}

$\Pi(3)=2$ {2,3}

$\Pi(4)=2$ {2,3}

$\Pi(5)=3$ {2,3,5}

92

Esercizio

- Si definisca il tipo di dato “numero complesso”, come una struttura `cplx` con campi `re` (parte reale) e `im` (parte immaginaria)
- Si scrivano due funzioni, che forniscono il modulo e l’angolo del numero complesso
- Funzioni utili (in `math.h`):
 - `atan` : arcotangente
 - `sqrt` : radice quadrata

