

# Fondamenti di Informatica

Allievi Automatici

A.A. 2015-16

Sottoprogrammi (Funzioni)

# Tanti problemi, una soluzione

- Abbiamo più volte incontrato blocchi di codice che risolvono particolari “sottoproblemi”:
- Come riutilizzare efficacemente tali blocchi?
- Come consentire ad altri di riutilizzarli?
- Come dare a questi blocchi un nome che ne indichi la funzionalità?
- Come “svincolare” lo sviluppo di soluzioni a sottoproblemi di questo tipo dallo sviluppo di una soluzione “complessa”?

# Analogia con la matematica

- Possiamo definire **funzioni**
  - Esempio: la funzione  $\min(X)$ 
    - $\min(X)$ : un valore  $m$  dell'insieme  $X$  tale che per ogni elemento  $w$  di  $X$  risulta  $m \leq w$ 
      - $\min(X) \triangleq \{ m \in X \mid \forall w \in X (w \geq m) \}$
- Una volta definite, possiamo usare le funzioni nei nostri ragionamenti tutte le volte che ci servono

# Motivazioni

- **Modularità** nello sviluppo del codice
  - Affrontare il problema per ***raffinamenti successivi***
- **Riusabilità**
  - Scrivere **una sola volta** il codice e usarlo più volte
  - Esempio: un algoritmo di ordinamento
- **Astrazione**
  - Esprimere in modo sintetico operazioni complesse
  - Definire **operazioni** specifiche dei tipi di dato definiti dal programmatore
    - Esempio: calcolo “*totale + iva*” di un ordine
    - **punti, segmenti, poligoni, numeri complessi**

# Astrazione

***“The purpose of abstraction is not to be vague,  
but to create a new semantic level  
in which one can be absolutely precise”***

E. W. Dijkstra, 1972

# Sulla **necessità** dei sottoprogrammi

I sottoprogrammi consentono di *scomporre* problemi complessi in moduli *più semplici*, sfruttabili poi, anche singolarmente, per la risoluzione di *problemi diversi*.

Se strutturati nel modo corretto, *nascondono* al resto del programma dettagli implementativi che non è necessario che esso conosca; in tal modo rendono *più chiaro* il programma nel suo complesso, e assai *più facile* la sua *manutenzione*

B. W. Kernighan

D. M. Ritchie

```
#define MaxNumStudenti 390;
```

```
typedef char string[20];
```

```
typedef struct {  
    int giorno, mese, anno; } data;
```

```
typedef struct {  
    int    matricola, voto;  
    string nome,  cognome;  
    data   dataNascita; } descrizioneStudente;
```

```
typedef struct {  
    int numStudenti;  
    descrizioneStudente iscritti[MaxNumStudenti]; } classe;
```

```
classe Fondamenti, Analisi;  
float mediaFondamenti, mediaAnalisi;
```

```
...
```

```
mediaFondamenti = valorMedio( Fondamenti );  
mediaAnalisi = valorMedio( Analisi );
```

**definizione di tipi**

**dichiarazione  
di variabili**

Calcolare il valor  
medio dei voti  
**di una classe**

# Come si fa in C?

- Dobbiamo definire un sottoprogramma che calcola “valorMedio”
  - Si tratta di un programma “asservito” al programma principale
- Il sottoprogramma deve essere **definito**
  - rispetto a dati generici (*una* generica classe di studenti)e poi può essere **chiamato** (o **invocato**)
  - cioè *attivato* per calcolare il risultato su dati particolari (di volta in volta, una *diversa* classe di studenti)
- Ogni sottoprogramma ha un **nome** e può produrre un valore come **risultato** della sua invocazione



# Il primo esempio

tipo del risultato  
della funzione

tipo dell'*argomento*  
della funzione

nome convenzionale  
della "generica" classe

testata

```
float valorMedio( classe c ) {  
    int i;  
    float m = 0.0;  
    for( i = 0; i < c.numStudenti; i++ )  
        m += c.iscritti[i].voto;  
    m = m / c.numStudenti;  
    return m;  
}
```

Nome della funzione

Valore restituito

## Un altro esempio

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

```
int power(int, int);
```

```
int main() {
```

```
    int i;
```

```
    for (i=0; i<10; i++)
```

```
        printf("%d %d %d\n", i, power(2,i), power(3,i));
```

```
    return 0;
```

```
}
```

dichiarazione  
della funzione

invocazioni  
della funzione

Variabili locali alla funzione

```
int power( int base, int n ) {
```

```
    int i, p=1;
```

```
    for( i=1; i<=n; i++ )
```

```
        p*=base;
```

```
    return p;
```

```
}
```

/\* base intera elevata a n intero \*/

definizione  
della funzione

Valore restituito

# ATTENZIONE ALLA i !!

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

```
int power(int, int);
```

```
int main() {  
    int i;  
    for (i=0; i<10; i++)  
        printf("%d %d %d\n", i, power(2,i), power(3,i));  
    return 0;  
}
```

La variabile i locale alla funzione power non aveva nessun rapporto con quella locale alla funzione main !!  
L'omonimia era casuale, possiamo rinominare una delle due variabili per enfatizzarne la totale indipendenza

```
int power( int base, int n ) {    /* base intera elevata a n intero */  
    int contatoreplp, p=1;  
    for( contatoreplp=1; contatoreplp<=n; contatoreplp++ )  
        p*=base;  
    return p;  
}
```

# I sottoprogrammi

- Quasi tutti i linguaggi di programmazione hanno una nozione di **sottoprogramma**
- Esistono sottoprogrammi che **restituiscono valori** e sottoprogrammi che non lo fanno
- Esempio:

```
double power( float val, int pow ) {  
    double result = 1.0;  
    int i;  
    for( i = 0; i < pow; i++ )  
        result = result * val;  
    return result;  
}
```

# Valori restituiti

- Chiamata al sottoprogramma precedente:

```
potenza = power( valore, esponente );
```

Il C suppone che tutti i sottoprogrammi restituiscano un valore. Se non si vuole che ciò accada, **occorre segnalarlo**

- Esempio di sottoprogramma **che non restituisce valori**:

```
void error_line( int line ) {  
    printf("Errore alla linea %d\n", line);  
}
```

- Tipica chiamata a un sottoprogramma che restituisce void:

```
error_line( line_number );
```

# Funzioni e Procedure

- Nella terminologia informatica:
  - i sottoprogrammi che restituiscono valori sono spesso chiamati **funzioni**
  - i sottoprogrammi che **non** restituiscono valori sono spesso chiamati **procedure**
- In C, però, sono chiamati tutti **funzioni**

parte dichiarativa **globale**

inclusione librerie /\* per poter richiamare funzioni utili (i/o, ...) \*/

**DICHIARAZIONE** di **variabili globali** e **funzioni**

**int main ( ) {**

parte dichiarativa **locale**

dichiarazione di **variabili locali**

istruzione 1; /\* tutti i tipi di operazioni, e cioè: \*/

istruzione 2; /\* istr. di assegnamento \*/

istruzione 3; /\* istr. di input / output \*/

...

istruzione N;

parte **esecutiva**

**}**

**DEFINIZIONE** delle funzioni

## Struttura di un programma C

# Dichiarazione delle funzioni

- È utile e raccomandato (standard ANSI C) riportare all'inizio del programma la “testata” (header) delle funzioni: **prototipi**
  - Nella parte dichiarativa globale o nel programma che chiama la funzione
- Serve a facilitare il lavoro del compilatore
  - In particolare del parser: analisi sintattica
- In pratica i prototipi delle funzioni si aggiungono alle dichiarazioni di variabili e costanti
- Specificano i nomi delle funzioni, i tipi restituiti e il *numero* e il *tipo* dei parametri che ricevono
  - Specificare il *nome* dei parametri, invece, è superfluo



# Funzioni e Procedure

```
void mess_err( void );

int main() {
    int a, b, c;
    printf("Dividendo: ");
    scanf("%d", &a);
    printf("Divisore: ");
    scanf("%d", &b);
    if( b != 0 ) {
        c = a / b;
        printf("%d diviso  
%d = %d", a, b, c);
    }
    else
        mess_err();
    return 0;
}
```

```
void mess_err( void ) {
    int i;
    char c;
    /*diverso dall'altro c*/

    for( i=0; i<=20; i++ )
        printf("\n");
    /* "pulizia" schermo */

    printf("ERRORE!  
DENOMINATORE NULLO!\n");
    printf("Premere un tasto  
per continuare.\n");
    scanf("%c", &c);
}
```

# Variabili locali e globali

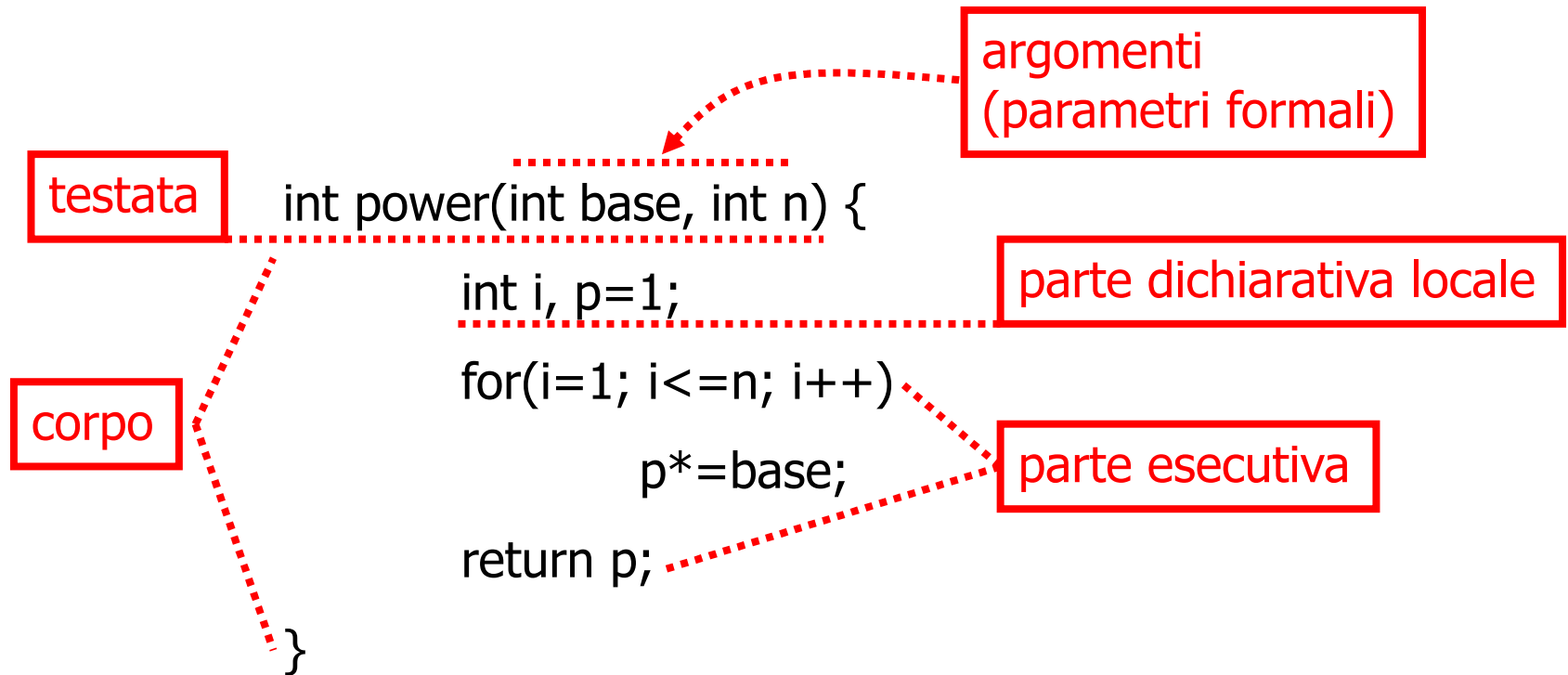
- Ogni sottoprogramma può usare **solo le variabili dichiarate al suo interno** più le variabili **globali**
- Le variabili locali nascono e muoiono con la vita del sottoprogramma (**lifetime**)
  - Dal momento in cui avviene il trasferimento del controllo fino al momento in cui il controllo è restituito al modulo “chiamante”
- Variabili non globali e non dichiarate all'interno di un sottoprogramma si possono usare solo **se passate come argomenti**

# Variabili locali e globali

- **Testata** (*function header*):
  - **tipo** del risultato (codominio della funzione)
  - **identificatore** del sottoprogramma
  - elenco delle dichiarazioni dei **parametri formali**
    - sono gli **argomenti** (il dominio!) della funzione, e ricevono un valore **nel momento in cui** la funzione viene attivata
- Parentesi graffe che racchiudono:
  - Parte dichiarativa **locale**
  - **Corpo** della funzione:

```
double power (float val, int pow) {  
    double ret_val = 1.0;  
    int i;  
    for (i = 0; i < pow; i++)  
        ret_val *= val;  
    return ret_val;  
}
```

# Esempio di definizione



# Corpo della funzione

- La parte esecutiva è costruita con le stesse regole del main (che, del resto, è una funzione)
- Contiene una o più istruzioni del tipo  
***return espressione;***
  - il **valore** dell'espressione diventa il risultato restituito
  - il **tipo** dell'espressione è quello indicato nella testata
  - **termina** l'esecuzione del sottoprogramma, restituisce il controllo al programma chiamante
  - visto dal programma chiamante, il valore restituito è il valore dell'espressione costituita dalla chiamata

## Alcune note

- In un sottoprogramma possono esserci nessuna o più istruzioni di *return*
  - ovviamente a ogni chiamata **se ne esegue solo una**
- Se l'istruzione *return* è assente, il sottoprogramma termina quando arriva alla } finale
- Il tipo speciale *void* viene usato per indicare il caso di assenza di valori di uscita o di parametri
- Le variabili dichiarate localmente “muoiono” all'uscita dal sottoprogramma

# Chiamata a sottoprogramma

- Ricordiamo:

`result = power( valore, esponente );`

- **Identificatore** della funzione
- Lista dei **parametri attuali** tra parentesi
  - **parametri attuali**: valori degli argomenti ai quali viene applicata la funzione
  - ogni parametro è un'espressione e può contenere altre chiamate di funzione
- Sintatticamente la chiamata è un'espressione
- **Corrispondenza posizionale**
  - al 1° corrisponde il 1°, al 2° il 2°, e così via ...

# Esempi di chiamate

- `x = sin(y) - cos(PI_GRECO - alfa);`  
*/\* PI\_GRECO definito come valore costante  $\pi$  \*/*
- `x = cos( atan(y) - beta );`
- `RisultatoDiGestione =`  
    `FatturatoTotale( ArchivioFatture ) -`  
    `SommaCosti( ArchivioCosti );`
- `OrdAlf = Precede( nome1, nome2 );`



# Un programma monolitico

```
int main() {
    char scelta;
    int val;
    while(1) { /* menu */
        printf("Premere A per inserire un numero tra 0 e 10 e calcolarne il cubo\n");
        printf("Premere B per inserire un numero tra 11 e 20 e calcolarne il quadrato\n");
        printf("Premere C per inserire un numero tra 21 e 30 e calcolarne il doppio\n");
        printf("Premere Q per uscire\n\n");
        scelta = getch();
        switch( scelta ) { /* scelte dell'utente */
            case 'a': case 'A': printf("Inserisci valore\n"); scanf("%d",&val);
                                printf("risultato=%d\n\n", val*val*val); break;
            case 'b': case 'B': printf("Inserisci valore\n"); scanf("%d",&val);
                                printf("risultato=%d\n\n", val*val); break;
            case 'c': case 'C': printf("Inserisci valore\n"); scanf("%d",&val);
                                printf("risultato=%d\n\n", val*2); break;
            case 'q': case 'Q': return 0;
            default: printf("Scelta non valida\n\n");
        }
    }
}
```

# Miglioriamolo...

```
void menu( void );
```

```
int main() {  
    char scelta;  
    int val;  
    while(1) {  
        menu();  
        scelta=getch();  
        ...  
        ...  
    }  
}
```

```
void menu( void ) {  
    printf("Premere A per inserire un numero tra 0 e 10 e calcolarne il cubo\n");  
    printf("Premere B per inserire un numero tra 11 e 20 e calcolarne il quadrato\n");  
    printf("Premere C per inserire un numero tra 21 e 30 e calcolarne il doppio\n");  
    printf("Premere Q per uscire\n\n");  
}
```

# Miglioriamolo ancora...

```
char menu( void );
```

```
int main() {  
    char scelta;  
    int val;  
    while(1) {  
        scelta = menu();  
        switch(scelta) {           /* scelte dell'utente */  
            ...  
        }  
    }  
}
```

```
char menu( void ) {  
    printf("Premere A per inserire un numero tra 0 e 10 e calcolarne il cubo\n");  
    printf("Premere B per inserire un numero tra 11 e 20 e calcolarne il quadrato\n");  
    printf("Premere C per inserire un numero tra 21 e 30 e calcolarne il doppio\n");  
    printf("Premere Q per uscire\n\n");  
    return getch();  
}
```

# Altro miglioramento...

```
char menu( void );
```

```
int leggi( void );
```

```
int main() {  
    char scelta; int val;  
    while(1) {  
        scelta = menu();  
        switch(scelta) {          /* scelte dell'utente */  
            case 'a': case 'A': val = leggi();  
                                printf("risultato=%d\n\n", val*val*val); break;  
            case 'b': case 'B': ...  
            ...  
        }  
    }
```

```
char menu( void ) { ... }
```

```
int leggi( void ) {  
    int v;  
    printf("Inserisci valore\n"); scanf("%d",&v); fflush(stdin);  
    return v;  
}
```

## E gli intervalli dei valori?

```
int leggi ( int min, int max ) {  
    int v, ok = 0;  
    while ( ! ok ) {  
        printf("Inserisci valore [%d,%d] : \n", min, max);  
        scanf("%d",&v); fflush(stdin);  
        if( v >= min && v <= max )  
            ok = 1;  
        else  
            printf(" Valore non compreso!\n");  
    }  
    return v;  
}
```

```
...  
case 'a': case 'A': val = leggi( 0, 10 );  
                    printf("risultato=%d\n\n", val*val*val); break;  
case 'b': case 'B': val = leggi( 11, 20 );  
                    printf("risultato=%d\n\n", val*val);      break;  
case 'c': case 'C': val = leggi( 21, 30 );  
                    printf("risultato=%d\n\n", val*2);          break;  
...
```

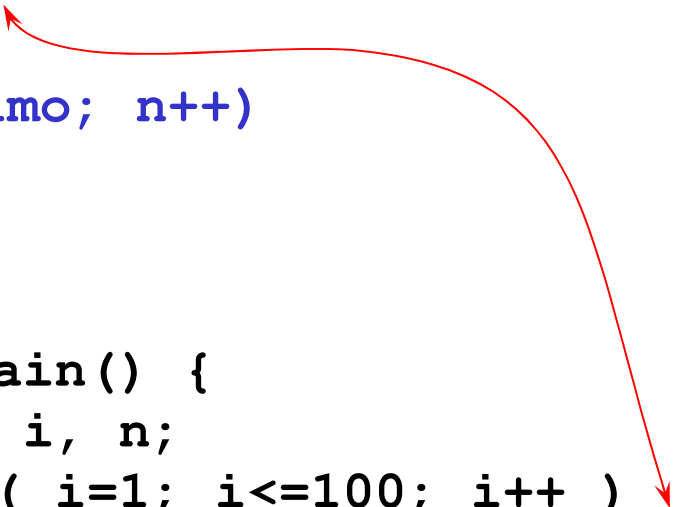
# Un programma poco leggibile

```
int main() {  
    int i,n, primo;  
  
    for( i=1; i<=100; i++ )  
    {  
        primo = 1;  
        for( n = 2; n<i/2 && primo; n++ )  
            if( i % n == 0 )  
                primo = 0;  
        if( primo )  
            printf("%i\n",i);  
    }  
    return 0;  
}
```

- La porzione di codice evidenziata calcola se il valore della variabile **i** è un numero primo
- Il valore su cui ‘lavora’ è il valore della variabile **i**
- Il suo ‘risultato’ è il valore della variabile **primo**

# La funzione verifica\_primo

```
int verifica_primo( int numero ) {  
    int n, primo = 1;  
    for( n=2; n<numero/2 && primo; n++)  
        if( numero % n == 0 )  
            primo = 0;  
    return primo;  
}  
  
int main() {  
    int i, n;  
    for( i=1; i<=100; i++ )  
        if( verifica_primo( i ) )  
            printf("%i\n", i);  
    return 0;  
}
```



# Parametri **formali** e **attuali**

- Nella **definizione** della funzione si dà un nome ai parametri (**formali**) che la funzione riceve in ingresso
  - In quel momento non vi è associato alcun valore

```
float power(float v, int e) { ... corpo ... }
```
  - Nella **dichiarazione** il nome non serve – basta specificare il tipo!

```
float power(float, int);
```
- Al momento della sua **invocazione** (chiamata) ad ogni parametro (**attuale**) corrisponde una *espressione*
  - La quale è valutata e ha un tipo e un valore

```
result = power( (strlen(s)-3)*val) , 2);
```
- Corrispondenza posizionale
  - Al primo corrisponde il primo, al secondo il secondo...
  - I tipi devono corrispondere (eventualmente c'è cast implicito)



# Esempio

Radice quadrata intera, cioè il max int il cui quadrato è  $\leq$  par

```
int radiceIntera( int par ) {  
    int cont;  
    cont = 0;  
    while( cont * cont <= par )  
        ++cont;  
    return cont - 1;  
    /* NB: quando si arriva qui, cont*cont > par */  
}
```

- **NB:** per tutti i valori di  $\text{par} < 0$  il risultato è sempre  $-1$   
Questo segnala un uso improprio della funzione  
(la radice dovrebbe essere... immaginaria?)


# Il perimetro rivisto con le funzioni

```
#include <math.h>
```

```
typedef struct { float x; float y; } punto;
```

```
float dist ( punto p1, punto p2 ) {  
    return sqrt( pow(p1.x-p2.x, 2) + pow(p1.y-p2.y, 2) );  
}
```

```
float perimetro ( punto poligono[], int dim ) {  
    int i; float perimetro = 0.0;  
    for ( i = 1; i < dim; i++ )  
        perimetro += dist(poligono[i-1], poligono[i]);  
    return perimetro + dist(poligono[0], poligono[dim-1]);  
}
```



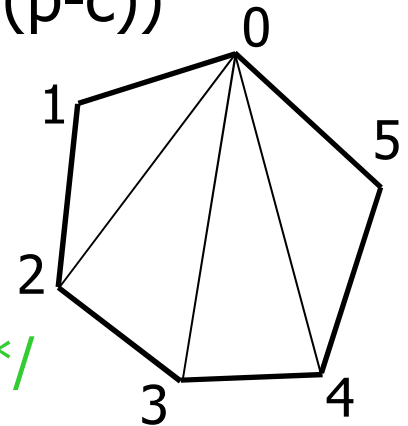
```
int main() {  
    punto pol[5];  
    ... acquisizione coordinate dei punti, omessa ...  
    printf("Il perimetro è %.2f", perimetro(pol, 5));  
    return 0;  
}
```

# Dal perimetro all'area

## Scomposizione del poligono in triangoli (a ventaglio)

Th. di Erone ( $\forall$  triangolo):  $A = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$

p: semiperimetro   a,b,c: lungh.lati   A: area



```
float erone( punto p1, punto p2, punto p3 ) {  
    punto tri[3]; float p;  
    tri[0]=p1; tri[1]=p2; tri[2]=p3; /* assegn. di struct */  
    p = 0.5 * perimetro(tri, 3);  
    return sqrt(p*(p-dist(p1, p2))*(p-dist(p2, p3))*(p-dist(p3, p1)));  
}
```

```
float areapol( punto polig[], int dim ) {  
    int i; float area = 0.0;  
    for ( i=2; i<dim; i++ )  
        area += erone( polig[0], polig[i-1], polig[i] );  
    return area;  
}
```

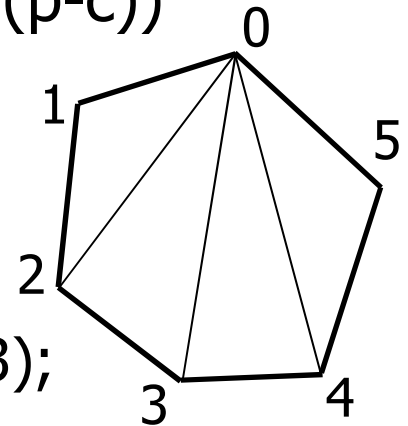
/\* N.B. Così funziona solo coi poligoni convessi \*/

## Dal perimetro all'area (2)

Scomposizione del poligono in triangoli (a ventaglio)

Th. di Erone ( $\forall$  triangolo):  $A = \sqrt{p \cdot (p-a) \cdot (p-b) \cdot (p-c)}$

p: semiperimetro   a,b,c: lungh.lati   A: area



```
float erone2( punto p1, punto p2, punto p3 ) {  
    float p, a, b, c;  
    a = dist(p1, p2); b = dist(p2, p3); c = dist(p1, p3);  
    p = (a+b+c) / 2;  
    return sqrt(p*(p-a)*(p-b)*(p-c));  
}
```

```
float areapol( punto polig[], int dim ) {  
    int i; float area = 0.0;  
    for ( i=2; i<dim; i++ )  
        area += erone2( polig[0], polig[i-1], polig[i] );  
    return area;  
}
```

*/\* N.B. Così funziona solo coi poligoni convessi \*/*

# Tipo del risultato e dei parametri

- **Tipo del risultato**
  - Può essere *built-in* o *user-defined*
  - **NON PUÒ essere un array**
    - **Ma può essere una struct**
      - Anche se tale struct contiene degli array!!!
  - **PUÒ** essere un puntatore a qualsiasi tipo
    - Ci torneremo studiando la memoria dinamica  
[*void \* malloc...*]

# Tipo del risultato e dei parametri

- **Tipo dei parametri**

- Può essere *built-in* o *user-defined*
- I parametri attuali **NON SONO MODIFICABILI** da parte delle funzioni a cui sono passati
  - I sottoprogrammi lavorano su copie dei parametri attuali
  - **Fanno eccezione (apparentemente) gli array**
    - La ragione è che si passano come parametri i "riferimenti alla prima cella", quindi in pratica delle copie di puntatori
    - Ma le copie dei puntatori sono altrettanto "valide" per modificare i parametri attuali !!

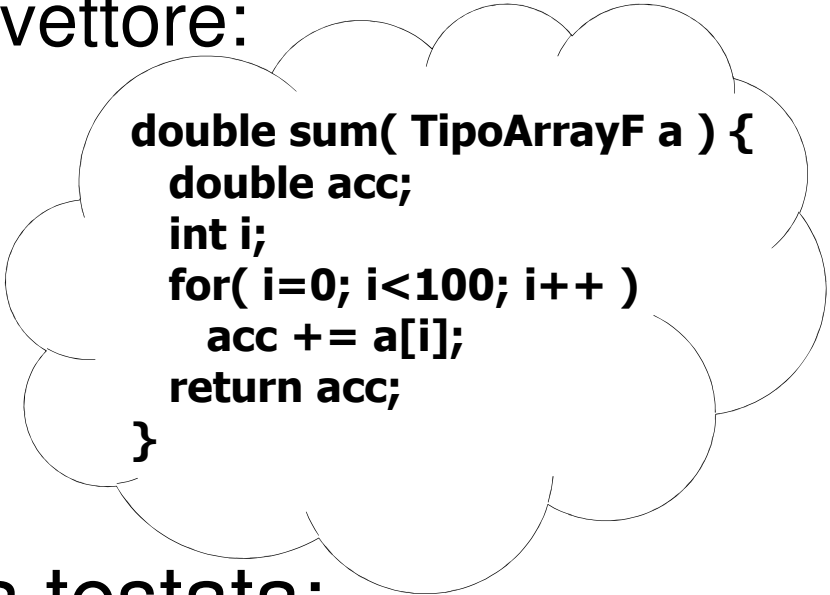
# Passaggio di array come parametri

- Parametro **attuale** di tipo array
  - si passa il valore dell'indirizzo base dell'array
    - È l'indirizzo della prima cella del primo elemento
  - N.B.: gli elementi dell'array NON sono copiati nel parametro formale
    - Solo l'indirizzo base viene copiato e passato
- Parametro **formale** di tipo array
  - rappresenta il puntatore al primo elemento

# Esempi

- Somma degli elementi di un vettore:

```
typedef double TipoArrayF[100];  
TipoArrayF pippo;  
...  
double ris;  
ris = sum( pippo );
```



```
double sum( TipoArrayF a ) {  
    double acc;  
    int i;  
    for( i=0; i<100; i++ )  
        acc += a[i];  
    return acc;  
}
```

- Sintassi equivalenti per la testata:

```
double sum( TipoArrayF a );  
double sum( double a[ ] );  
double sum( double * a );
```



```

#include <stdio.h>

void modificaInt(int);
void modificaArray(int[]);

int main() {
    int a = 0, i, v[4] = {0,1,2,3};

    printf(" a = %d\n", a);    /* valori prima */
    for ( i = 0; i < 4; i++ )
        printf(" v[%d] = %d\n", i, v[i]);

    modificaInt( a );
    modificaArray( v );

    printf(" a = %d\n", a );    /* valori dopo */
    for ( i = 0; i < 4; i++ )
        printf(" v[%d] = %d\n", i, v[i]);

    return 0;
}

```

```

void modificaInt( int i ) {
    i += 100;
}

void modificaArray( int vett[] ) {
    int i;
    for ( i = 0; i < 4; i++ )
        vett[ i ] += 100;
}

```

***Che cosa stampa il programma?***

## strcmp(s1, s2)

- E se la funzione non ci fosse già?
  - controlliamo un carattere alla volta
  - interrompiamo il controllo appena sono diverse

```
int mystrcmp( char s1[ ], char s2[ ] ) {  
    int i = 0;  
    while ( s1[i] == s2[i] && s1[i] != '\0' )  
        ++i;  
    return s1[i] - s2[i];  
}
```

```
int mystrcmp2( char * s1, char * s2 ) {  
    while (*s1 == *s2 && *s1 != '\0')  
        { s1++; s2++; }  
    return *s1 - *s2  
}  /* ...versione più... "acrobatica"! */  
   /* però... meno leggibile */
```

# Variabili “collettive” - confronto

- **ARRAY** (variabili “omogenee”)
  - **Non si possono** assegnare collettivamente
  - Quando sono passati come parametri a una funzione, **possono** esserne modificati
  - **Non possono** essere restituiti tramite *return*
- **STRUCT** (variabili “eterogenee”)
  - **Si possono** assegnare collettivamente
  - Quando sono passate come parametri a una funzione, **non possono** esserne modificate
  - **Possono** essere restituite tramite *return*

```
typedef int array [SIZE];  /* Parte dichiarativa globale */
```

```
typedef struct { int actualSize; /* Dich. di tipo */  
                array contents;  
            } tabella;
```

```
int search(tabella, int); /* Dich. di funzione */
```

```
int main() {  
    int i, pos, val;  
    tabella myTab;  
    printf("dammi dimensione della tabella (inferiore a %d) ", SIZE);  
    scanf("%d", &myTab.actualSize); /*manca il controllo actualSize < size*/  
    for(i=0; i<myTab.actualSize; i++) {  
        printf("dammi un valore della tabella ");  
        scanf("%d", &myTab.contents[i]);  
    }  
    printf("dammi valore da cercare in tabella ");  
    scanf("%d", &val);  
    pos = search(myTab, val);  
    if (pos!=-1)  
        printf("elemento trovato in posizione %d \n", pos);  
    else  
        printf("valore non trovato in tabella\n");  
    return 0;  
}
```

```
int search (tabella t, int k) {  
    int n;  
    for( n = 0; n < t.actualSize; n++ )  
        if (t.contents[n]==k)  
            return n;  
    return -1;  
}
```

# Sintassi dell'invocazione

- Una funzione può essere vista come un'*espressione*, che può essere *valutata*
  - possiamo pensare una funzione **somma** (**a**, **b**) equivalente a **a+b**
- Quando, nell'esecuzione del codice, è necessario 'valutare' un'espressione-funzione, la funzione viene *invocata*
- La sintassi per la formulazione di una espressione-funzione è semplicemente:  
<nome funzione>(<lista argomenti>),
- dove <lista argomenti> è una lista di espressioni di tipo corrispondente a quelle della definizione

# Semantica dell'invocazione

## Invocazione di una funzione:

1. Le espressioni che ne costituiscono gli argomenti vengono valutate, e il valore reso disponibile alla funzione
2. La funzione viene eseguita, fino a quando non viene incontrato un comando return
3. Il “valore” dell'espressione-funzione è il valore dell'argomento del return che ha causato la terminazione

# Funzioni: modello di esecuzione

- Immaginiamo che esista **una macchina dedicata** a eseguire la funzione **main()**
- Immaginiamo che sia **ogni volta creata** una nuova e diversa macchina **dedicata** ad eseguire ciascuna funzione, all'atto della sua **chiamata**
- Ogni macchina dedicata a una funzione ha una sua **memoria**, per le variabili locali, per i valori dei parametri che ricevono e per il risultato che restituiscono
  - Tale memoria si dice anche **ambiente** della funzione

# Passaggio Parametri

Approfondiamo il meccanismo  
di comunicazione dei dati  
dal programma chiamante  
al sottoprogramma chiamato



# Modifica dei parametri attuali

- Tutti i parametri, in C, sono passati per **COPIA**
- Le variabili passate come parametri a una funzione, quindi, se alterate nell'ambiente della funzione, **non cambiano valore** nell'ambiente del chiamante (parametri passati *per valore*)
- Se si vuole che una funzione agisca sulle variabili dell'ambiente del chiamante, occorre passare **l'indirizzo** di tali variabili (parametri passati *per locazione* o *per indirizzo*)  $\Rightarrow$  uso dei **puntatori**
- Diciamo che **modifichiamo i parametri attuali**

# Modifica dei parametri attuali

- Nella **definizione** il parametro formale deve essere di tipo *puntatore* al tipo del parametro attuale di cui si vuole la modifica
- Nella **chiamata** si deve passare l'indirizzo (usando &) del parametro attuale da modificare  
**NOTA:** il parametro attuale **deve** essere una **variabile** (o una espressione che, valutata, restituisca un puntatore); **NON** una generica **espressione**
- Nel corpo della funzione si usa l'operatore \* di *dereferenziazione* per riferirsi al parametro

## Scambio dei valori di due interi (versione scorretta)

```
void swap( int p, int q ) {  
    int temp; /* var. locale */  
    temp = p;  
    p = q;  
    q = temp;  
}
```

**Chiamata: swap(i, j);**

**ERRORE: Alla fine i e j sono immutate!!!**

# Scambio dei valori di due interi

```
void swap(int * p, int * q) {  
    int temp; /* variab. locale */  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

**Chiamata:** `swap(&i, &j);`

# Puntatori e funzioni

```
void fiddle( int x, int * y ) {  
    printf("Begin fiddle: x=%d, y=%d\n", x, *y);  
    x = x+1;  
    *y = *y+1; Due parametri: x (int) e y (punt. a int)  
    printf("End fiddle:    x=%d, y=%d\n", x, *y);  
}  
  
int main ( ) {  
    int i = 0, j = 0;  
    printf("Begin main:    i=%d, j=%d\n", i, j);  
    fiddle(i, &j);  
    printf("Returned, ending main: i=%d, j=%d\n", i, j);  
}
```

N.B.: si “crea” un puntatore

usando & all'interno della chiamata `fiddle (i, &j);`

# Traccia di esecuzione

Begin main:    i=0, j=0

Begin fiddle: x=0, y=0

End fiddle:    x=1, y=1

Returned, Ending main: i=0, j=1

- All'uscita da fiddle il valore di i è *rimasto lo stesso*, mentre quello di j (passato tramite puntatore) è *cambiato*
- Se vogliamo poter modificare il valore di una variabile in modo che *resti modificato* all'uscita dalla funzione, occorre *passarla tramite un puntatore*

# Intercambiabilità di procedure e funzioni

## ⇒ Funzione:

```
int f( int par1 ) {  
    ... (calcola valore  
della variabile  
"risultato") ...  
    return risultato;  
}
```

chiamata:

```
y = f(x);
```

## ⇒ Procedura:

```
void f( int par1, int * par2 ) {  
    ... (calcola valore di  
variabile "risultato")  
    ...  
    *par2 = risultato;  
}
```

chiamata:

```
f(x, &y);
```

a questo punto a y è stato  
assegnato il valore

# Raccomandazioni di stile

- Per le funzioni:
  - passare i parametri per valore
  - non tentare di accedere a variabili non locali
- Una funzione che abbia parametri passati per puntatore e ne modifichi il valore, si dice avere un *effetto collaterale* (side effect)
- Benché in sé siano ammissibili, gli effetti collaterali si devono usare con *prudenza*, e in generale sono *sconsigliabili* quando non sono strettamente necessari



# Parametri: riassumiamo pro e contro

<div> <div>Modi di passaggio</div> <div>caratteristiche</div> </div>	<b>per valore</b> ("per copia")	<b>per indirizzo</b> ("per riferimento") ("per puntatore")
<b>Tempo e spazio</b> necessari a trasferire (copiare) i dati	<b>grandi</b> (per parametri di grandi dimensioni)	<b>piccoli</b> (la dimensione dei puntatori è fissa, non dipende dai dati)
C'è rischio di <b>effetti collaterali</b> indesiderati?	<b>No</b> (i parametri attuale e formale sono distinti)	<b>Sì</b> (i parametri attuale e formale "di fatto" coincidono)
Permette la <b>restituzione</b> di <b>valori</b> al chiamante?	<b>No</b>	<b>Sì</b>

# Parametri di tipo array e struct

- Per passare a una funzione un parametro di tipo array occorre passarne l'indirizzo base, perciò di fatto **gli array sono sempre passati per indirizzo**
  - Analogamente, una funzione **non può restituire un array** (inteso come "il suo contenuto"), ma solo **un puntatore a un array** (cioè il puntatore al suo primo elemento)
- Un parametro di tipo struct si può passare **sia per indirizzo sia per valore** (anche se la struct contiene campi di tipo array!)
  - Analogamente, una funzione **può restituire una struct** (anche se la struct contiene degli array)

# Parametri di tipo array

## Altre particolarità degli array con le funzioni:

- Esempio: `typedef double TipoArray [DIMENSIONE];`
- Prototipi equivalenti: `double sum( TipoArray a, int n )`  
`double sum( double a[], int n )`  
`double sum( double *a, int n )`
- N.B.: non si deve specificare la dimensione del vettore; n rappresenta la porzione dell'array da considerare valida
- La funzione somma i primi n elementi di un array.  
Supponiamo di avere un array `V[50]`:

`sum(V, 50);` restituisce `v[0] + v[1] + ... v[49]`

`sum(V, 30);` restituisce `v[0] + v[1] + ... v[29]`

`sum(&V[5], 7);` restituisce `v[5] + v[6] + ... v[11]`

`sum(V+5, 7);` restituisce `v[5] + v[6] + ... v[11]`

# Somma dei primi n elementi di un array di double

```
/* n-1 rappresenta la posizione occupata  
dell'ultimo elemento dell'array che contiene un  
valore significativo (la "coda" da n+1 a  
DIMENSIONE non è considerata significativa) */  
  
double sum( double a[], int n ) {  
    int i;  
    double ris = 0.0;  
    for( i = 0; i < n; i++ )  
        ris += a[i];  
    return ris;  
}
```

# Ancora sui parametri di tipo array

## Per gli array mono-dimensionali:

- Dichiariamo i parametri formali come puntatori al tipo degli elementi dell'array
- Dichiarazione del parametro formale

```
... f( UnTipo vet[] )
```

- Non occorre, sintatticamente, specificare la dimensione statica degli array: il compilatore può eseguire il calcolo dello spiazzamento in base al tipo puntato
  - UnTipo v[N];
  - $v[i] \equiv *(v+i)$ 
    - serve conoscere solo `sizeof(UnTipo)`, N non serve
- Dichiarando (**correttamente**) in alternativa:

```
... f( UnTipo * vet )
```

il compilatore **può** comunque risolvere l'espressione `vet[i]`

# Ancora sui parametri di tipo array

## Per gli array multi-dimensionali

- Per calcolare lo spiazzamento occorrono alcune dimensioni intermedie
  - `UnTipo m[X][Y], c[X][Y][Z];`
  - $m[i][j] \equiv *((*(m+i)+j) \approx *((*m + Y*i + j) \approx *(m[0]+ Y*i + j) \approx *(&m[0][0]+ Y*i + j)$ 
    - serve conoscere `sizeof(Tipo)` e **Y** (**X non serve**)
  - $c[i][j][k] \equiv *((*(c+i)+j)+k) \approx *((**c + Y*Z*i + j*Z + k) \approx *(c[0][0]+Y*Z*i+j*Z+k) \approx *(&c[0][0][0] + Y*Z*i + j*Z + k)$ 
    - servono `sizeof(Tipo)`, **Y** e **Z** (**X non serve**)
- Sintatticamente: occorre specificare tutte le dimensioni meno l'ultima
- Dichiarazioni corrette:  
`... g(UnTipo mat[][Y]) ... h(UnTipo cube[][Y][Z])`
- Dichiarando (**erroneamente**):  
`... g(UnTipo * mat) ... h(UnTipo * cube)`

il compilatore **non può** risolvere le espressioni `mat[i][j]` e `cube[i][j][k]`

# Rivediamo la copia di stringhe

- Copia con sovrascrittura
  - Copia di un carattere alla volta fino a '\0' (incluso)
  - **ATTENZIONE:** la memoria per s1 deve già essere stata esplicitamente allocata [array di dimensioni sufficienti]


```
void strcpy( char s1[ ], char s2[ ] ) {  
    int i = 0;  
    if( s1 != NULL && s2 != NULL ) {  
        while ( s2[i] != '\0' ) {  
            s1[i] = s2[i];  
            i++;  
        }  
        s1[i] = '\0';  
    }  
}
```

```
void strcpy2( char s1[ ], char s2[ ] ) {  
    while( s1 && s2 && ( *(s1++)=*(s2++) ) != '\0' )  
        ;  
} /* ...una versione assai più... "acrobatica"! */  
/* "Bella", però... è codice "poco leggibile" */
```

# Ordinamento di array

## — bubblesort —

```
void bubblesort( int param[], int size ) {  
    int i, j;  
    for( i=0; i<size-1; i++ )  
        porta in posizione i il minimo in [i ... size-1]  
}
```



```
for ( j=size-1 ; j>i ; j-- )  
    if ( param[j] < param[j-1] )  
        swap( param+j, param+j-1 );
```

Attenzione: si usa la funzione **swap**  
passandole **gli indirizzi** degli elementi da scambiare



# Un altro problema di “modellazione”

Gestione dei numeri complessi  
(un **tipo di dato astratto**)

- Fornire un’astrazione completa dei numeri complessi mediante **dati** e **operazioni**
- Prescindere dall’implementazione (trasparenza delle scelte realizzative)

```
#include <math.h> /* sqrt e fabs (radice e valore ass. di un float) */
```

```
typedef struct { float re; float im; } nC;
```

```
void printC( nC z ) { /* a+bi è visualizzato come "± |a| ± |b| i " */  
    char sre = '+', sim = '+';  
    if ( z.re < 0.0 )  
        sre = '-';  
    if ( z.im < 0.0 )  
        sim = '-';  
    printf(" %c %.2f %c %.2f i ", sre, fabs(z.re), sim, fabs(z.im));  
}
```

```
float modulo( nC z ) {  
    return sqrt( z.re*z.re + z.im*z.im );  
}
```

```
nC costruisci( float r, float i ) { /* restituisce una struct di tipo nC */  
    nC temp;  
    temp.re = r;  
    temp.im = i;  
    return temp;  
}
```

```
nC quadratoC( nC z ) { /* versione “funzionale” */  
    return costruisci( z.re * z.re - z.im * z.im , 2 * z.re * z.im )  
}
```

```
void quadratoC2( nC z, nC *ris ) { /* versione “procedurale” */  
    ris->re = z.re * z.re - z.im * z.im;  
    ris->im = 2 * z.re * z.im;  
}
```

```
void leggiC( nC *z ) {  
    printf("Parte reale : ");  
    scanf("%f", &(z->re));  
    printf("Parte immaginaria : ");  
    scanf("%f", &(z->im));  
}
```

*/\* versione “procedurale” \*/*

```
nC leggiC2() {  
    nC temp;  
    printf("Parte reale : ");  
    scanf( "%f", &(temp.re));  
    ...  
    return temp;  
}
```

*/\* versione “funzionale” \*/*

```

nC somma( nC z, nC w ) {    /* restituisce z+w */
    return costruisci( z.re+w.re, z.im+w.im );
}

```

```

nC prodotto( nC z, nC w ) {    /* restituisce z*w */
    return costruisci( z.re * w.re - z.im * w.im,
                       z.re * w.im + z.im * w.re );
}

```

```

nC quoziente( nC z, nC w ) {    /* restituisce z/w */
    return costruisci (
        (z.re*w.re + z.im*w.im) / (w.re*w.re + w.im*w.im) ,
        (z.im*w.re - z.re*w.im) / (w.re*w.re + w.im*w.im)
    );
}

```

```
nC inverso( nC z ) {                                     /* restituisce 1/z */  
    return quoziente( costruisci(1.0, 0.0), z );  
}
```

```
nC potenzaIntera( nC b, int e ) {                         /* restituisce  $b^e$  */  
    nC result = costruisci(1.0, 0.0);  
    if ( e < 0 ) {  
        e = - e;  
        b = inverso( b );  
    }  
    for( ; e > 0; e-- )  
        result = prodotto( result, b );  
    return result;  
}
```