

Sottoprogrammi e compilazione separata

Prof. Salvatore Venticinque

Prof. Mario Magliulo

-
- Quando si scrive un programma succede spesso che si debba eseguire numerose volte una stessa sequenza di istruzioni, sugli stessi dati o su dati diversi.
 - Per evitare di riscrivere più volte queste sequenze di istruzioni e per limitare le dimensioni dei programmi, il linguaggio C dispone di una particolare struttura chiamata ***funzione***

Funzioni

- Una **funzione** è un modulo di programma la cui esecuzione può essere invocata più volte nel corso del programma principale.
- La funzione può calcolare uno o più valori, che saranno comunicati al programma chiamante al termine della sua esecuzione, oppure può eseguire azioni generiche (come, ad esempio, l'aggiornamento della base dati).
- Il nome **funzione** deriva dall'evidente analogia con le funzioni intese in senso matematico.
- Poiché la **funzione** fornisce un valore, ad essa è associato un *tipo*.

Approccio top-down

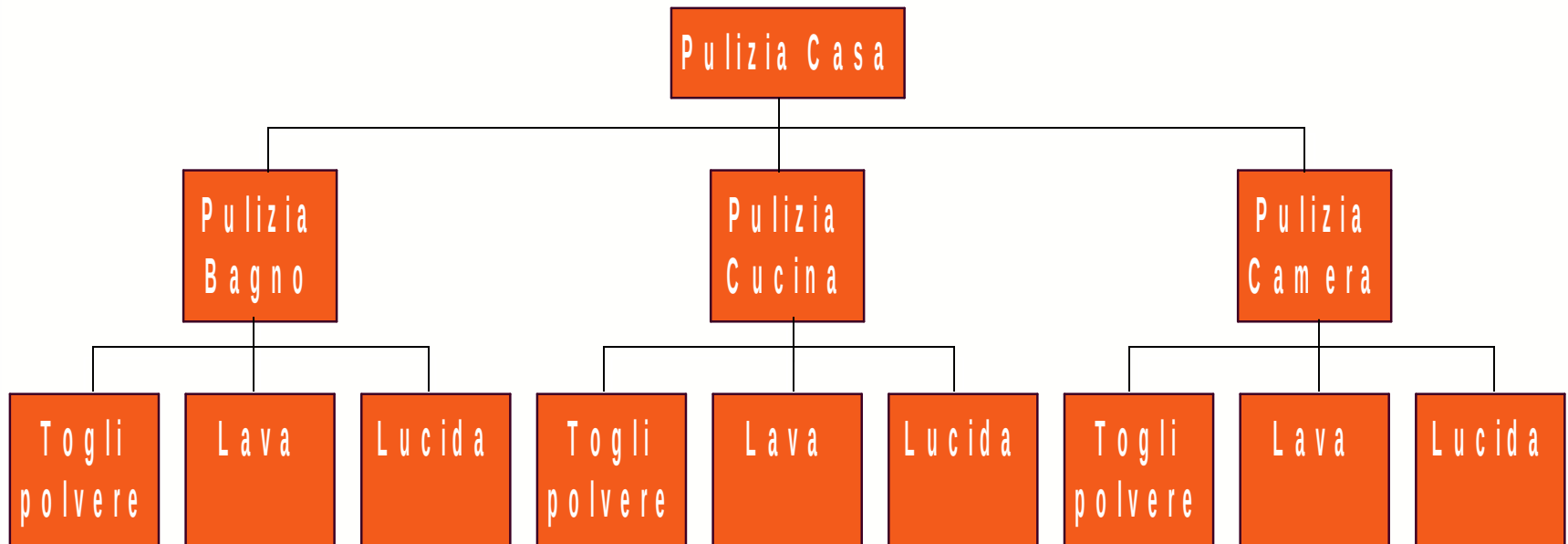
- Altra giustificazione all'esistenza dei sottoprogrammi deriva dal fatto che un programma può consistere di decine di migliaia di istruzioni.
- Sebbene fattibile, una soluzione "monolitica" del problema è inefficiente in quanto complicata da "*debuggare* " e difficile da leggere.
- Per questo si tende a organizzare il programma complessivo in "*moduli* " ognuno dei quali tratta e risolve solo una parte del problema.
- In altre parole si adotta, per la soluzione del problema, il cosiddetto "approccio *top-down* "

Approccio gerarchico

- Il problema è decomposto in una sequenza di sottoproblemi più semplici.
- Quest'azione è ripetibile su più livelli, ovvero ogni sottoproblema può a sua volta essere decomposto in una sequenza di sottoproblemi.
- La decomposizione prosegue sino a che si ritiene che la sequenza è composta ormai solamente da “*sottoproblemi terminali*”, ovvero da sottoproblemi risolvibili in modo “semplice”.

Approccio top-down

Esempio: pulizia di una casa



Paradigma procedurale

- I linguaggi di programmazione permettono di suddividere le operazioni in modo simile tramite sottoprogrammi.
- La sequenza dei sottoproblemi terminali si traduce in una sequenza di sottoprogrammi.
- Così pure, poiché una *funzione* può invocare un'altra *funzione*, la gerarchia delle operazioni si traduce in una gerarchia di sottoprogrammi.
- Genericamente si chiamano *procedure* i sottoprogrammi che NON ritornano un risultato, *funzioni* i sottoprogrammi che ritornano un risultato (di qualche tipo primitivo o non).

Sottoprogrammi

E' possibile aggregare gruppi di istruzioni per formare dei “semilavorati” detti sottoprogrammi.

Vantaggi:

- evitare di replicare porzioni di codice sorgente
- Invocando un sottoprogramma si manda in esecuzione la porzione di codice corrispondente.
- consentono al programmatore di avere tante chiamate ma una sola porzione di codice.
- Una raccolta di sottoprogrammi può essere organizzata in librerie
- Una libreria può essere utilizzata senza conoscerne i dettagli implementativi.

Es: `printf()` e `scanf()`, la cui dichiarazione è contenuta nel file `stdio.h`.

Funzioni

- In C i sottoprogrammi sono detti funzioni:
- a partire da uno o più valori presi in ingresso, esse ritornano un valore al programma chiamante.



Dichiarazione di una funzione

tipo_ritorno nome_funz (tipo_par1, ..., tipo_parN);

- *Tipo di ritorno*
- *Nome funzione*
- *Parentesi*
- *Lista dei parametri: tipo [nome]*

Se il tipo è void la funzione viene solitamente detta
procedura

Esempio

- `double cubo(float c);`
- `float somma (float a, float b);`
- `void stampat(int v[100]);`

Definizione funzione

tipo_ritorno nome_funz (tipo_par1 par1, ..., tipo_parN parN)

{

...

}

Esempio

```
double cubo(double c){  
    double temp;  
    temp = c*c*c;  
    return temp;  
}
```

```
void stampa(int v[100])  
{  
    for(int i=0;i<100;i++)  
        printf("%d ",v[i]);  
}
```

Parametri

I parametri dichiarati nella definizione:

- Sono detti ***parametri formali***
- È importante l'ordine
- E' importante il tipo
- E' importante riconoscere se sono di ingresso, uscita o ingresso-uscita

Overload

Funzioni diverse possono distinguersi:

- Per il nome
- Per il numero dei parametri
- Per l'ordine dei parametri

```
float somma(float a, float b);
```

```
float somma(float a, float b, float c);
```

```
float max(float a, float b);
```

```
double somma(float a, float b);
```

```
double somma(float c, float b);
```

Funzione potenza

//dichiarazione

```
float pote(float, int);
```

//definizione

```
float pote(float b, int e)
```

```
{
```

```
    float temp = 1;
```

```
    for(int i=0;i<e;i++)
```

```
        temp = temp*b;
```

```
    return temp;
```

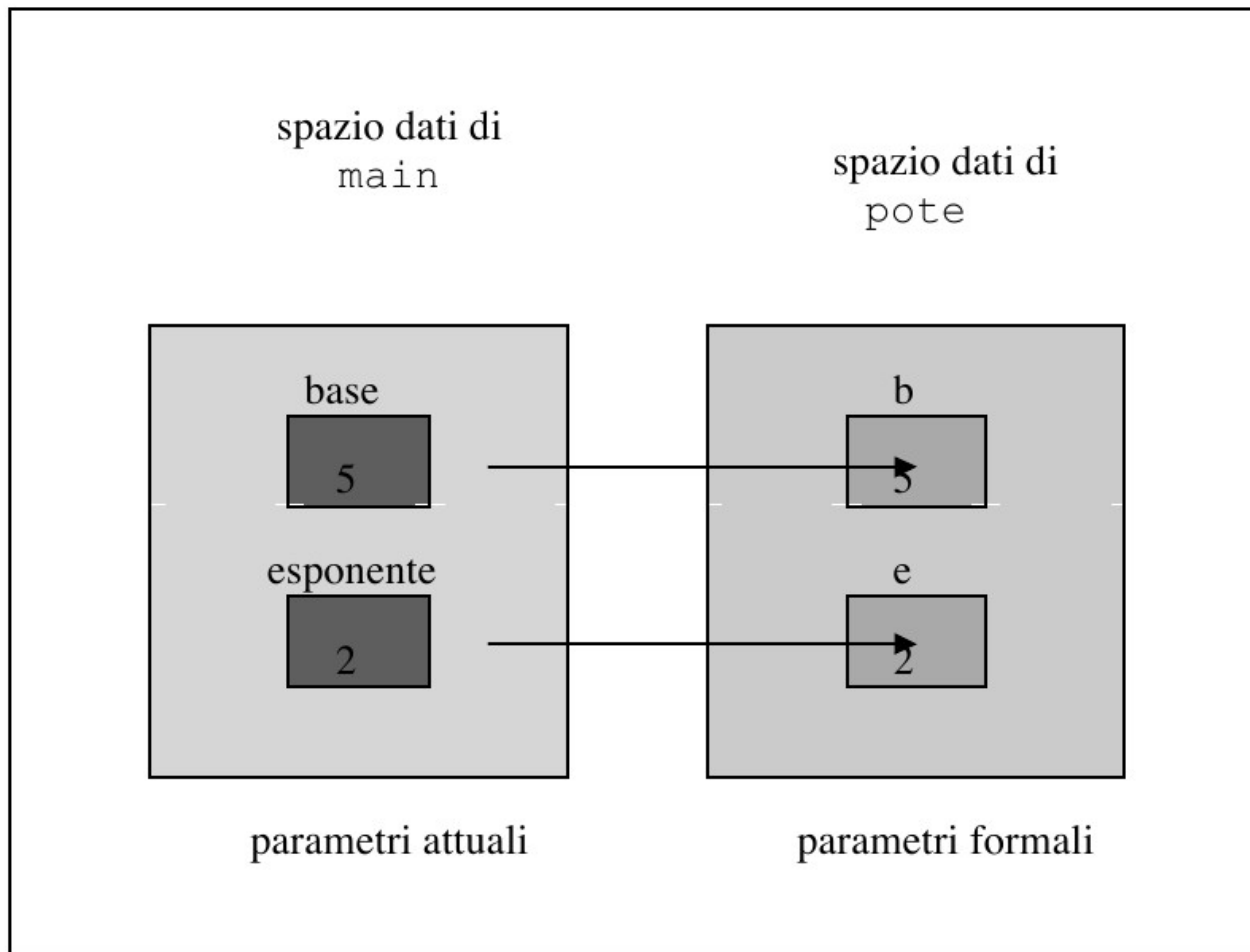
```
}
```


Chiamata di una funzione

```
int main()
{
    float risultato, base = 5;
    int esponente = 4;
    risultato = pote(base,esponente);
    printf("%f", risultato);
    printf("%f", pote(3,2));
}
```

- Occorrono passare i ***parametri effettivi***
- Stesso tipo della dichiarazione
- Stesso ordine della dichiarazione
- Nome anche diverso

Allocazione memoria e visibilità variabili



Parametri formali e variabili locali

Quando una funzione viene chiamata:

- 1) Viene allocato spazio per i parametri formali
- 2) Viene compiuto il valore dei parametri effettivi nei parametri formali
- 3) Vengono allocato spazio per le variabili locali
- 4) Viene eseguita la funzione
- 5) Viene deallocato lo spazio delle variabili locali
- 6) Viene deallocato lo spazio dei parametri formali
- 7) Si procede con l'esecuzione del programma chiamante

Osservazioni

- Alla funzione vengono passati i valori
- Tutti i cambiamenti su variabili locali e parametri vengono persi
- Può andare bene per i parametri di ingresso ...
- ... per i parametri di uscita?

Variabili globali e locali

```
char s[]="ciao";
int lunghezza( );
int main()
{
    printf("%s\n",s);
    printf("%d\n",lunghezza());
}
```

```
int lunghezza( ){
    int i=0;
    while(s[i]!='\0')
        i++;
    return i;
}
```

- Il main è una funzione come le altre
- Non usare, se non assolutamente necessario, variabili globali

Parametri di uscita: il caso del vettore

- Stampa di un vettore:
- Parametri di ingresso?
- Parametri di uscita?

Parametri di uscita: il caso del vettore

Dichiarazione:

- `void stampa(int v[], int n);`
- `void stampa(int v[100], int n);`

Chiamata:

- Qual'è valore viene passato?
- Cosa comporta?

```
int main(){  
    int vettore[100];  
    [...]  
    stampa(vettore, 100);  
}
```

Lettura degli elementi del vettore

- Parametri di ingresso?
- Parametri di uscita?

Prototipo

Dichiarazione:

```
int leggiv(int v[],int n);
```

Definizione:

```
int leggiv(int v[], int n)
```

```
{  
    int r;  
    do{  
        scanf("%d",&r);  
    }while(r<0||r>n);  
    for(int i=0;i<r;i++)  
        scanf("%d",&v[i]);  
    return r;  
}
```

Passaggio di parametri per indirizzo

- Più parametri di uscita?
- Es: scambio di due variabili?

```
void swap(int a , int b)
```

```
{ int temp;
```

```
    temp = a;
```

```
    a=b;
```

```
    b=temp;
```

```
}
```

Problema??

Servono i puntatori ...

Passaggio parametri per indirizzo

Esempio sbagliato per lo scambio dei parametri:

```
void scambia (int x, int y){  
    int tmp;  
    tmp = x;  
    x = y;  
    y = tmp;  
}
```

Passaggio valori

```
int a = 4, b = 5;  
scambia(a,b);
```

stack

&a

4

&b

5

Frame
chiamante

&x

4

&y

5

Frame
scambia

Scambio valori nella funzione

```
int a = 4, b = 5;  
scambia(a,b);
```

&a	4
&b	5

Frame
chiamante

&x	5
&y	4

Frame
scambia

Effetto della chiamata

```
int a = 4, b = 5;  
scambia(a,b) ;  
printf("%d,%d",a,b) ;
```

&a

4

&b

5

Frame
chiamante

Ricapitolando

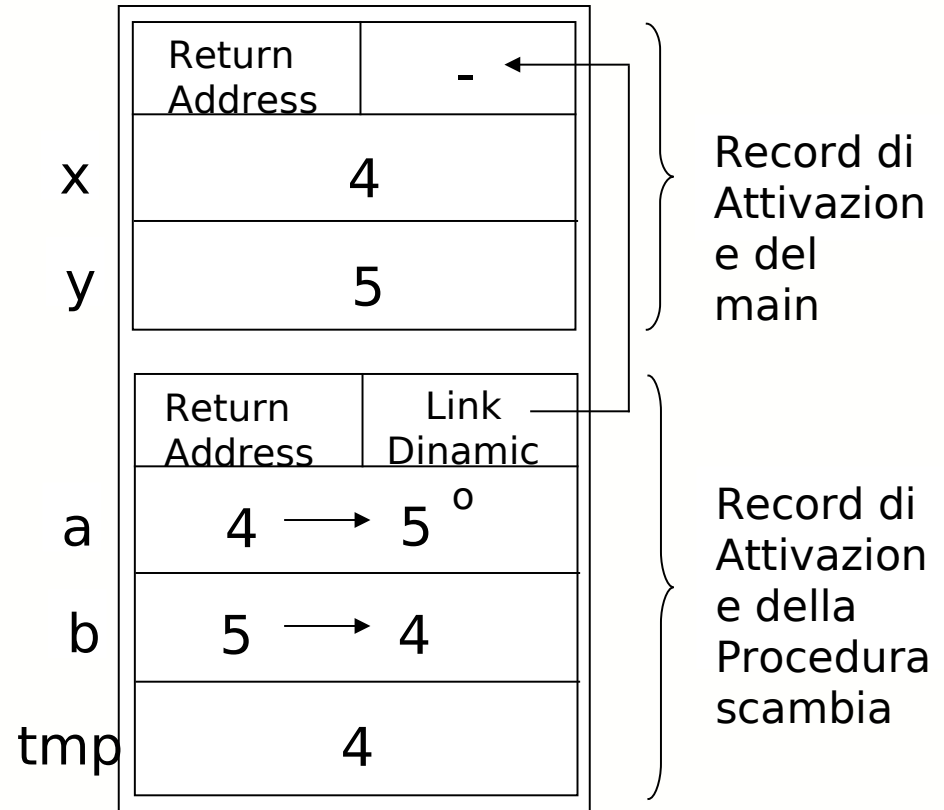
```
#include <stdio.h>

void scambia ( int , int );

main(){
    int x=4, y=5;

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

void scambia ( int a, int b){
    int tmp=a;
    a=b;
    b=tmp;
}
```



Versione corretta dello scambio

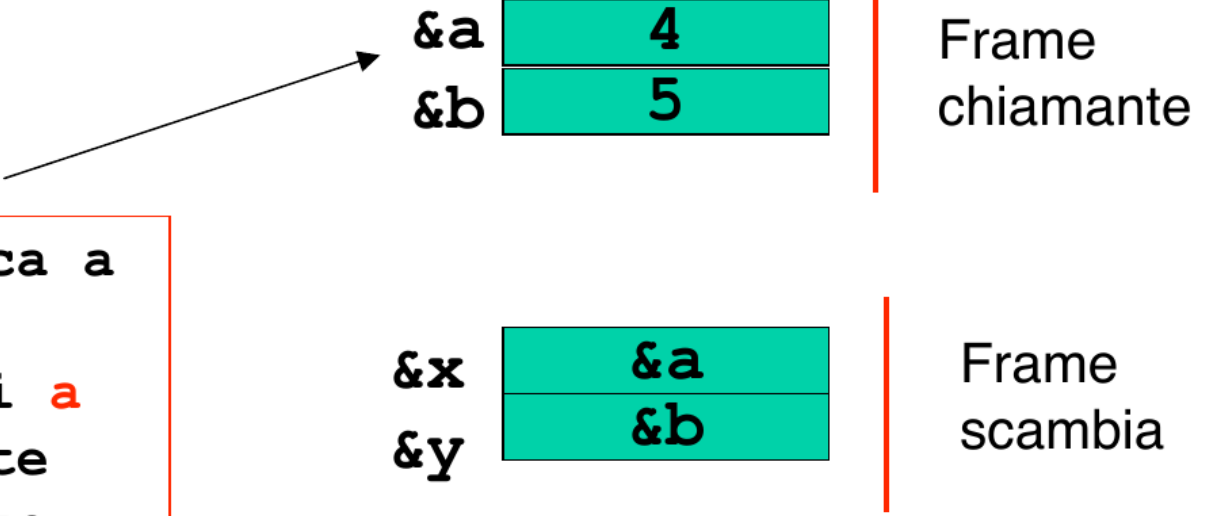
```
void scambia (int *x, int *y){  
    int tmp;  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}
```

- con chiamata `scambia(&a,&b)`

Passaggio valori

```
int a = 4, b = 5;  
scambia(&a, &b);
```

stack



&a	4
&b	5

Frame
chiamante

Ogni modifica a
***x** modifica
il valore di **a**
nell'ambiente
del chiamante

&x	&a
&y	&b

Frame
scambia

Scambio nella funzione

```
int a = 4, b = 5;  
scambia(&a, &b);
```

&a	5
&b	4

Frame
chiamante

&x	&a
&y	&b

Frame
scambia

Effetto dello scambio

```
int a = 4, b = 5;  
scambia(&a, &b);  
printf("%d, %d", a, b);
```

&a

5

&b

4

Frame
chiamante

Ricapitolando

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

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

```
main(){
```

```
    int x=4,y=5;
```

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

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

```
}
```

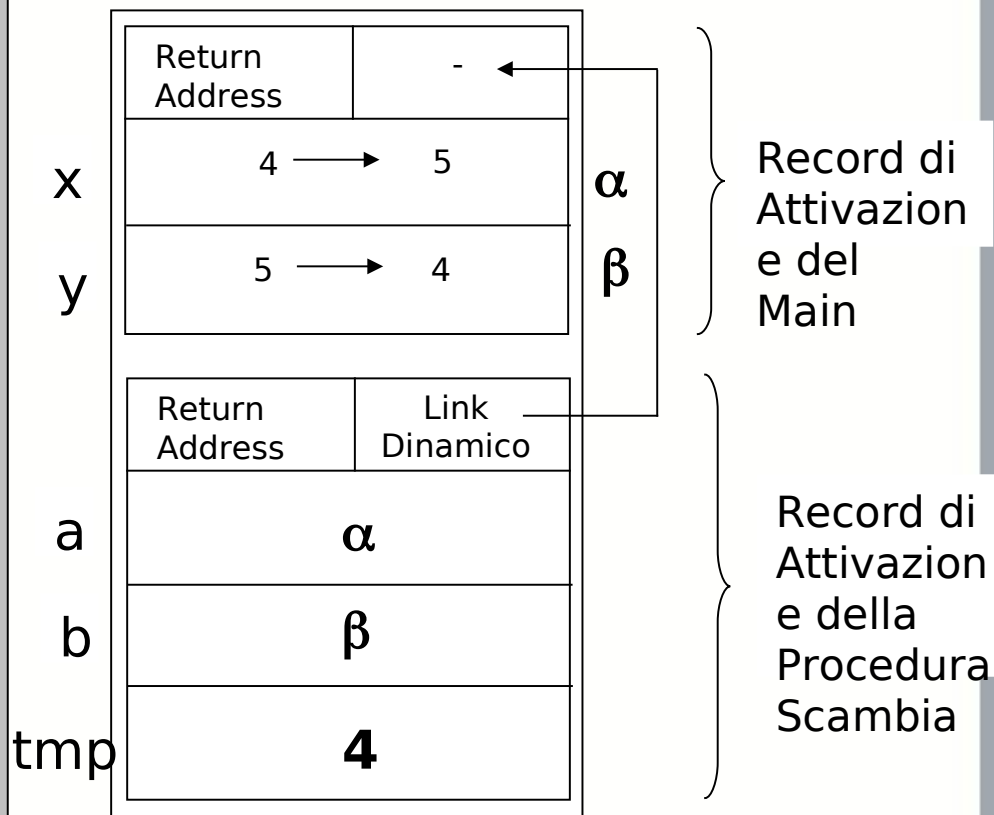
```
void scambia ( int *a, int *b) {
```

```
    int tmp=*a;
```

```
    *a=*b;
```

```
    *b=tmp;
```

```
}
```



Massimo e minimo vettore

```
void maxmin(int v[],int n, int * max,int* min)
{
    *max = v[0];
    *min = v[0];
    for(int i=1;i<n;i++)
    {
        if(*max<v[i]) *max=v[i];
        if(*min>v[i]) *min=v[i];
    }
}
```

```
int main()
{
    int v[] = {1,4,2,5,3,7,9,0};
    int n = 8; int a,b;
    maxmin(v,n,&a,&b);
    printf("max: %d, min: %d\n",a,b)
}
```

Organizzazione del codice in librerie

Organizzare Librerie

- File header
 - Dichiarazione dei prototipi
- File .c
 - Definizione delle funzioni
- File .o
 - Libreria compilata

Esempio: myarray.h

- `int length(char s[]),`

myarray.c

```
int length(char s[])
{
    int i=0;
    while(s[i]!='\0') i++;
    return i;
}
```

Programma principale

#include <stdio.h>



#include "myarray.h"



Dove cercare le librerie?

```
int main()
```

```
{
```

```
    char stringa[] = "salvatore";
```

```
    printf("lunghezza: %d\n", length(stringa));
```

```
}
```

Compilazione libreria

`gcc -c myarray.c → myarray.o`

`gcc -c programma.c → programma.o`

`gcc -o programma programma.o myarray.o`

Cosa succede se manca il file .h nella cartella?

Cosa succede se eseguo:

`gcc -o programma programma.o`