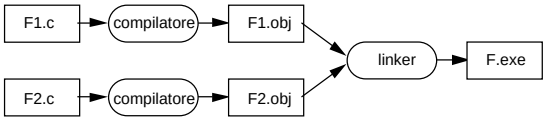
# APPLICAZIONI SU PIU' FILE

**Serve poter sviluppare applicazioni su piú file:**
- **alcune funzioni e alcune definizioni di dati in un file**
- **altre funzioni e dati in file diversi**

**Perché??**
**1. Se il programma è di dimensioni notevoli:**
- **piú facile scrivere e aggiornare file piccoli**
- **divisione logica tra le varie perti del programma si riflette a livello di divisione fisica tra i file**
- **modifiche chiaramente localizzate**
- **migliora i tempi di compilazione (quando si fanno delle modifiche si ri-compilano solo i file modificati e poi si rifa il link) → concetto di progetto!**

**2. Se il programma è scritto da un team:**
- **ognuno scrive su propri file e poi si collega il tutto**




**Cosa serve?**
- **I programmi devono usare dati e funzioni definiti altrove (in altri file)!**
- **Una metodologia per scomporre le cose su piú file**

**Vediamo quindi questi due punti:**
- **gestione variabili**
- **metodologia di scomposizione**



# CLASSI DI MEMORIZZAZIONE
## TEMPO di VITA — VISIBILITÀ

In C, ogni **entità (variabile o funzione)** usata in un programma è caratterizzata da
- **Nome**, identificatore unico nel programma
- **Tipo**, per indicare l'insieme dei valori
- **Valore**, tra quelli ammessi dal tipo
- **Indirizzo**, riferimento alla memoria che la contiene
- **Tempo di vita**, durata di esistenza nel programma
- **Visibilità** (scope) del nome nel programma

Tempo di vita e visibilità sono specificati mediante la **CLASSE di MEMORIZZAZIONE** ⇨ indica il tipo di area di memoria in cui una entità viene memorizzata
**NOTA BENE:** In altri linguaggi, tempo di vita e visibilità di una entità non sono concetti indipendenti uno dall'altro

Le classi di memorizzazione sono 4:
1. **auto** ⇨ *automatica*
2. **register** ⇨ registro (*caso particolare di auto)*
3. **static** ⇨ *statica*
4. **extern** ⇨ *esterna*

**IMPORTANTE:** La classe di memorizzazione può essere applicata alla **definizione** sia di **variabili** che di **funzioni**
**PERÒ ...** per **variabili** sono applicabili tutte e 4
per **funzioni** sono applicabili solo **static** e **extern**
Alle **dichiarazioni** si applica**, in genere,** solo la classe di memorizzazione **extern**



## CLASSI DI MEMORIZZAZIONE PER LE VARIABILI

**VISIBILITÀ**
possibilità di riferire la variabile
**TEMPO di VITA**
durata della variabile all'interno del programma

### 1. CLASSE di MEMORIZZAZIONE auto

- **default** per **variabili locali a un blocco o funzione**
  **NOTA BENE:** non si applica alle funzioni
- **VISIBILITÀ**
  La variabile è **locale** e quindi, è visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi
- **TEMPO DI VITA**
  la variabile è **temporanea** cioé esiste dal momento della definizione, sino all'uscita dal blocco o dalla funzione in cui è stata definita
- **ALLOCAZIONE:**
  su **STACK** (valore iniziale indefinito di default)

**ESEMPIO:**

```
somma(int v[ ],int n)
{
int k,sum = 0;      /*  Quanto vale k ?  */
/* è come scrivere auto int k,sum = 0; */
for (k = 0; k < n; k++) sum += v[k];
return sum;
}
```



### 2. CLASSE di MEMORIZZAZIONE register

- **come le auto**
  ***e quindi***
  **NOTA BENE:** non si applica alle funzioni

  **VISIBILITÀ**
  La variabile è **locale** e quindi, è visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi
  **TEMPO DI VITA**
  la variabile è **temporanea** cioé esiste dal momento della definizione, sino all'uscita dal blocco o dalla funzione in cui è stata definita
- **ALLOCAZIONE:**
  su **REGISTRO MACCHINA** (valore iniziale indefinito di default)
  Solo se possibile cioè se:
  - registri disponibili
  - dimensione variabile compatibile con quella dei registri

**ESEMPIO:**

```
somma(int v[ ],int n)
{
register int k,sum = 0;
for (k = 0; k < n; k++) sum += v[k];
return sum;
}
```

**NOTA:**
La classe di memorizzazione register può essere usata anche per i parametri di una funzione

### 3. CLASSE di MEMORIZZAZIONE static

- **TEMPO DI VITA**
  la variabile è **permanente** per tutto il programma: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine

La definizione di una variabile statica può essere:
1. globale cioé esterna ad ogni funzione *oppure*
2. locale cioè all'interno di una funzione o blocco

- QUESTO INFLUENZA LA **VISIBILITÀ**
  1. la variabile è visibile ovunque, dal punto di definizione in poi, ma **solo all'interno del file che la contiene**
  2. visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi
- **ALLOCAZIONE:**
  nei DATI GLOBALI (***valore iniziale di default 0***)
  Per il microprocessore 8086/88 l'allocazione è nel DATA SEGMENT

**ESEMPIO:**

File "CCC.c"

```
fun1(...);
funA(void);
extern funB(void);
static int ncall = 0;
...
static fun1(...)
  { ncall++; ... }
funA(void)
  { return ncall; }
```

File "DDD.c"

```
void fun1(...);
funB(void);
extern funA(void);
static int ncall = 0;
...
static void fun1(...)
  { ncall++; ... }
funB(void)
  { return ncall; }
```



### ESEMPI: VARIABILI AUTOMATICHE E STATICHE

**ESEMPIO 1: Variabile statica locale**

```
#include <stdio.h>
void static_demo (void);   /* dichiarazione funzione */
main()
{      int i;

for( i= 0; i < 10; ++i)
       static_demo();
/* ...- static_variable ... ERRORE!!! */
}

void static_demo(void)
{      int variable = 0;
       static int static_variable;
printf("automatic = %d, static = %d\n",
       ++variable, ++static_variable);
}
```

`variable` ### ***variable automatica***
**visibile** solo nella funzione `static_demo()` e con
**tempo di vita** pari alla singola invocazione
**allocata** nella parte di STACK e
**inizializzata** esplicitamente sempre a 0 ad ogni invocazione

`static_variable` ### ***variabile statica locale***
**visibile** solo nella funzione `static_demo()`, ma con
**tempo di vita** pari a tutto il programma
**allocata** nella parte di DATI GLOBALI e
**inizializzata** implicitamente a 0 solo all'inizio dell'esecuzione del programma

Quindi il valore della variabile `variable` è sempre uguale ad 1, mentre il valore della variabile `static_variable` viene incrementato ad ogni chiamata



**ESEMPIO 2: Variabile statica globale**

```
/* file static1.c */
#include <stdio.h>

static int static_var;
void st_demo (void);/* dichiarazione funzione */

void main()
{      int i;
for( i= 0; i < 10; ++i) st_demo();

static_var = 100;
/* printf("automatic = %d\n", variable);  ERRORE!!! */
printf("static globale = %d\n", static_var);
}

void st_demo(void)
{      int variable = 0;

printf("automatic = %d, static globale = %d\n",
       ++variable, ++static_var);
}
```

`variable` ### ***variable automatica***
⇨ *come prima*

`static_var` ### ***variabile statica globale***
**visibile** solo nel file `static1.c`
**tempo di vita** pari a tutto il programma
**allocata** nella parte di DATI GLOBALI e
**inizializzata** implicitamente a 0 solo all'inizio dell'esecuzione del programma

Quindi il valore della variabile `variable` è sempre uguale ad 1, mentre il valore della variabile `static_var` viene incrementato ad ogni chiamata e poi viene posto uguale a 100 nella funzione main()



### 4. CLASSE di MEMORIZZAZIONE extern

- **default** per **variabili globali** cioé esterne ad ogni funzione
- vale sia per definizioni che per dichiarazioni
- **VISIBILITÀ**
  **globale** cioé la variabile è visibile ovunque, dal punto di definizione in poi anche **al di fuori del file** che ne contiene la definizione
- **TEMPO DI VITA**
  la variabile è **permanente** per tutto il programma: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine
- **ALLOCAZIONE:**
  nei DATI GLOBALI (***valore iniziale di default 0***)
  Per il microprocessore 8086/88 l'allocazione è nel DATA SEGMENT

**ESEMPIO:**

File "AAA.c"

```
extern void fun2(...)
...
int ncall = 0;
...
fun1(...)
{
ncall++;
...
}
```

File "BBB.c"

```
extern fun1(...);
void fun2(...);
...
extern int ncall;
...
void fun2(...)
{
ncall++;
...
}
```

**ESEMPIO: VARIABILE EXTERN**

```
/* file main.c */
#include <stdio.h>
int var;
/* definizione variabile esterna: extern di default */
extern void demo (void);
/* dichiarazione funzione esterna */

void main()
{      int i;
for( i= 0; i < 10; ++i)
       demo();
var = 100;
/* printf("automatic = %d\n", variable);  ERRORE!!! */
printf("extern = %d\n", var);
}
```

```
/* file demo.c */
#include <stdio.h>

extern var;   /* dichiarazione variabile esterna*/
void demo(void)
{      int variable = 0;
printf("automatic = %d, extern = %d\n",
       ++variable, ++var);
}
```

**variable** viene posta sullo STACK e inizializzata a 0 **ad ogni invocazione della funzione** demo
**var** viene posta nella parte DATI GLOBALI e inizializzata a 0 **una sola volta**

**ANSI C:**
`int var;` ⇨ viene considerata una **definizione** perchè **non** è stata usata esplicitamente la classe di memorizzazione **extern** (valida di default)
`extern var;` ⇨ viene considerata una **dichiarazione** perchè è stata usata esplicitamente la classe di memorizzazione **extern** (valida di default)



**CLASSI DI MEMORIZZAZIONE PER LE FUNZIONI**

**VISIBILITÀ**
possibilità di riferire la funzione
**TEMPO di VITA**
durata della funzione all'interno del programma
⇨ **sempre globale** cioé pari all'intera durata del programma

**ALLOCAZIONE:**
sempre nella parte di CODICE
Per il microprocessore 8086/88 l'allocazione è nel CODE SEGMENT

**NOTA BENE:**
Le classi di memorizzazione auto, register e static locali (a blocchi o funzioni) non hanno senso poiché NON è possibile definire una funzione all'interno di un'altra funzione (o blocco)

**1. CLASSE di MEMORIZZAZIONE static**

La definizione di una funzione statica può essere solo globale cioé esterna ad ogni funzione

- **VISIBILITÀ**
  la funzione è visibile ovunque, dal punto di definizione in poi, ma **solo all'interno del file che la contiene**



**ESEMPIO:**

File "CCC.c"

```
fun1(...);
funA(void);
extern funB(void);
static int ncall = 0;
...
static fun1(...)
  { ncall++; ... }
funA(void)
  { return ncall; }
```

File "DDD.c"

```
void fun1(...);
funB(void);
extern funA(void);
static int ncall = 0;
...
static void fun1(...)
  { ncall++; ... }
funB(void)
  { return ncall; }
```

**ESEMPIO: FUNZIONE STATICA**

```
#include <stdio.h>
static void static_fun (void);
/* dichiarazione funzione */
void main()
{      int i;
for( i= 0; i < 10; ++i)
       static_fun();
}
static void static_fun(void)
{ printf("Sono una funzione statica:
             sono visibile solo in questo file\n");
}

static void static_fun(void);
```

***dichiarazione/prototipo funzione***

⇨ questa dichiarazione serve per poter usare questa funzione nel main(), riportando la definizione alla fine
Si può evitare, se si definisce direttamente la funzione prima del main()
La classe di memoria `static` può anche essere omessa

`static void static_fun(void) {...};` ***definizione funzione***



**2. CLASSE di MEMORIZZAZIONE extern**

- **default** per le **funzioni**
- vale sia per definizioni che per dichiarazioni
- **VISIBILITÀ**
  **globale** cioé la funzione è visibile ovunque, dal punto di definizione in poi anche **al di fuori del file** che ne contiene la definizione

**ESEMPIO:**

File "AAA.c"

```
extern void fun2(...)
...
int ncall = 0;
...
fun1(...)
{
ncall++;
...
}
```

File "BBB.c"

```
extern fun1(...);
void fun2(...);
...
extern int ncall;
...
void fun2(...)
{
ncall++;
...
}
```

**ESEMPIO: FUNZIONE EXTERN**

**NOTA BENE: è lo stesso di prima**

```
/* file main.c */
#include <stdio.h>
int var;
/* definizione variabile esterna: extern di default */

extern void demo (void);
/* dichiarazione funzione esterna */

void main()
{      int i;
for( i= 0; i < 10; ++i)
       demo();
var = 100;
/* printf("automatic = %d\n", variable);  ERRORE!!! */
printf("extern = %d\n", var);
}
```

```
/* file demo.c */
#include <stdio.h>

extern var;   /* dichiarazione variabile esterna*/

void demo(void)
/* definizione funzione esterna: extern di default */
{      int variable = 0;
printf("automatic = %d, extern = %d\n",
       ++variable, ++var);
}
```

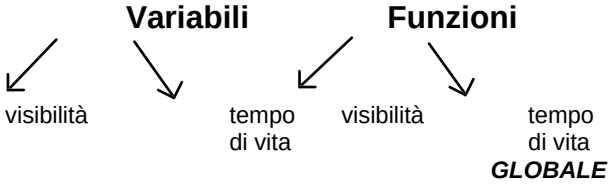
`extern void demo (void);` ***dichiarazione/prototipo funzione***

⇨ si usa la stessa convenzione usata per le variabili anche se per una funzione la differenza fra definizione e dichiarazione è sempre chiara

`void demo (void) { ... };` ***definizione funzione***



# CLASSI DI MEMORIA DELLE ENTITÀ IN C




## CLASSI DI MEMORIZZAZIONE

**1) auto** ⎫ ⎬ **register** ⎭ **N.B.: solo per variabili**
### variabili LOCALI a blocchi o funzioni (default auto)
**VISIBILITÀ: limitata al blocco**
**TEMPO DI VITA: limitata al blocco**
**ALLOCAZIONE: STACK** (auto)

**2) static** ⇨ **visibilità e tempo di vita scorrelato**
a) dentro a funzioni o blocchi
**N.B.:** solo per variabili
**VISIBILITÀ: limitata al blocco**
**TEMPO DI VITA: globale**
b) fuori da qualunque funzione
sia variabili che funzioni
**VISIBILITÀ: limitata al file**
**TEMPO DI VITA: globale**
**ALLOCAZIONE: DATI GLOBALI** (variabili)
**CODICE** (funzioni)

**3) extern** ⇨ **variabili e funzioni**
(default a livello di file)
**VISIBILITÀ: globale**
**TEMPO DI VITA: globale**
**ALLOCAZIONE: DATI GLOBALI** (variabili)
**CODICE** (funzioni)



# APPLICAZIONE SU PIÙ FILE

La presenza di **definizioni** e **dichiarazioni** di entità insieme con il concetto di **classe di memoria** rende possibile sviluppare una applicazione su più file

Ogni singolo file viene **compilato** in modo **INDIPENDENTE** e poi i vari file oggetto sono messi insieme al **collegamento**

In un singolo file, per poter usare entità definite negli altri file è necessario **dichiarare le entità esterne** utilizzate
- Infatti, durante la compilazione di un singolo file sorgente, il compilatore non può conoscere le entità (variabili e funzioni) definite negli altri file e quindi ha necessità delle loro dichiarazioni per poter fare gli opportuni controlli che il loro uso sia appropriato
- **è necessario dichiarare le entità esterne utilizzate**

**DICHIARAZIONE**: specifica le proprietà di una entità
- sia **funzione** (in ANSI C mediante il suo prototipo)
- sia **variabile**
- sia **tipo di dato**

➠ in ogni modo, non viene allocato spazio in memoria

```
extern fattoriale(int n);  /* prototipo funzione */
extern float xyz;   /* dichiarazione variabile */
typedef short int Signed16;/* dichiarazione tipo */
```

**DEFINIZIONE**: specifica le proprietà di una entità e la sua allocazione
- sia **funzione**
- sia **variabile**

```
fattoriale(int n) {.../* codice funzione */}
int xyz = 10.5;
```



segue Applicazione su più file

Ogni entità può essere dichiarata *più volte* (in file diversi) ma deve essere definita *una e una sola* volta

Una entità è **dichiarata nei file** che la usano
ma
**definita solo ed unicamente in un file** che la alloca

Sia per le dichiarazioni che per la definizione si deve usare la classe di memoria **extern**

La clausola extern quindi è usata
sia da chi le esporta (cioè chi mette a disposizione l'entità),
sia da chi la importa (cioè chi usa l'entità),
seppure con semantica diversa

La classe **extern** è il **default** per ogni entità definita/dichiarata a livello di programma

## METODOLOGIA DI USO

(adottata dall'ANSI C)

una sola *definizione* (con eventuale inizializzazione esplicita) in cui non compare esplicitamente la clausola extern
le *dichiarazioni* riportano esplicitamente la classe *extern*

*Esempio di Programma Contenuto in più File*

InfoBase.c;

```
#include <stdio.h>
#define Null ' ' /* elemento particolare */
typedef char Atomo;
int Stato=0;
void Acquisisci(Atomo *A);
void Visualizza(Atomo A);
void Acquisisci(Atomo *A){
  scanf("%c", &A);
  if (A==Null)
        Stato=5;
  } /* end Acquisisci */
void Visualizza(Atomo A){
   printf("%c",A);
} /* end  Visualizza /*
```

ProvaInfoBase.c;

```
typedef char Atomo; /*Occorre di nuovo Atomo*/
extern int Stato;
extern void Acquisisci(Atomo *A);
extern void Visualizza(Atomo A);

main() {
  Atomo MioAtomo;
  Acquisisci(&MioAtomo);
  if (Stato==0) ...;
  ...
}
```

Se si toglie `extern` nella dichiarazione `extern int Stato;` la compilazione va a buon fine ma in fase di link si ottiene un errore in quanto `Stato` risulta essere definita sia in `ProvaInfoBase.c` che in `InfoBase.c` infatti, senza `extern`, cioè `int Stato;` è considerata una definizione

Se invece tolgo `extern` in `extern void Acquisisci(Atomo *A);` sia la compilazione che il link vanno a buon fine infatti, senza `extern`, cioè con `void Acquisisci(Atomo *A);` ottengo ancora una dichiarazione della funzione non una definizione.



**ESEMPIO:**

Il file "f3.c" **mette a disposizione**
la variabile x e la funzione f() - DEFINIZIONI

I file "f1.c" e "f2.c" **utilizzano** la variabile x e la funzione f()
messa a disposizione dal file "f3.c" - DICHIARAZIONI

f1.c

```
extern  int x;


extern float f
       (char c);

/*dichiarazioni
==> IMPORT */

void prova()
{
< uso di x e f >
}
```

f2.c

```
extern  int x;


extern float f
       (char c);

/*dichiarazioni
==> IMPORT */

void main()
{
< uso di x e f >
}
```

f3.c

```
int x = 10;

float f (char c);
{  var  locali  e
codice di f >
}

/*definizioni
==> EXPORT
*/
```

**COMPILAZIONE INDIPENDENTE**
bisogna **compilare** f1.c, f2.c e f3.c

**LINKING**
bisogna fare il **linking** di f1.obj, f2.obj e f3.obj **insieme**
➠ **RISOLVE I RIFERIMENTI ESTERNI**

per ottenere il programma nella sua forma eseguibile



segue ESEMPIO:

Tutte le **dichiarazioni** possono essere inserite **in un HEADER FILE** "f3.h" incluso dai file utilizzatori. Serve per:
- non riscrivere un sacco di volte le stesse dichiarazioni su piu' file
- per modificarle una sola volta le dichiarazioni e fare avere effetto a queste modifiche su tutti i file cui servono le dichiarazioni

"f3.h"

```
Extern int x;
extern float f(char c);
   ...
```

f1.c

```
#include "f3.h"

/*dichiarazioni
==> IMPORT */

void prova()
{
< uso di x e f >
}
```

f2.c

```
#include "f3.h"

/*dichiarazioni
==> IMPORT */

void main()
{
< uso di x e f >
}
```

f3.c

```
int x = 10;

float f (char c);
{  var  locali  e
codice di f >
}

/*definizioni
==> EXPORT
*/
```

Un **header file** contiene *solitamente* **dichiarazioni** e non **definizioni**
➠ vedi file header di libreria



## Struttura di un programma (in generale)

***In ogni file,*** possiamo avere

| **DICHIARAZIONE** di | **DEFINIZIONE** di |
|---|---|
| Tipi<br>Variabili<br>Funzioni | Variabili (Dati)<br>Funzioni (Algoritmi) |

- Ogni programma, anche se suddiviso su più file, deve contenere *sempre* **una**, ed **una sola**, **funzione** di nome **main**
- L'esecuzione avviene attraverso **funzioni che si invocano**

  ***la visibilità da un file all'altro viene garantita dalle dichiarazioni extern di variabili/funzioni definite extern di default***
  - l'esecuzione inizia dalla funzione **main**
  - il main può invocare altre funzioni (anche di altri file)
  - l'esecuzione termina quando
    - termina il flusso di esecuzione del main
    - viene chiamata una delle funzioni di sistema che fanno terminare l'esecuzione (ad es. **exit**)
- Le variabili possono essere usate (sono visibili) solo **dopo** la loro definizione o dichiarazione di tipo **extern**
- Le funzioni possono essere usate anche **prima** della loro definizione, purchè vengano dichiarate

  nel caso che siano definite in altri file, la dichiarazione deve presentare esplicitamente la classe **extern**

## Struttura di un programma (ogni singolo file)

** inclusione **header file** per **librerie standard** C
#include <stdio.h> ...
** dichiarazione **tipi**
... tipo1; ... ... tipoN;
** definizione **variabili globali** all'intero programma
tipoVar1 nomeVar1, ...; ...;
tipoVarJ nomeVarJ, ...;
** definizione **variabili statiche**
**static** tipoVarJ+1 nomeVarJ+1, ...; **static** ...;
**static** tipoVarK nomeVarK, ...;
** dichiarazione **variabili globali** all'intero programma
**extern** tipoVarK+1 nomeVarK+1, ...; **extern** ...;
**extern** tipoVarN nomeVarN, ...;
** dichiarazione **prototipi funzioni** (definite sotto)
tipo1 F1(parametri); ... **static** tipoK+1 FK+1(parametri); ...
tipoK FK(parametri); **static** tipoJ FJ(parametri);
** dichiarazione **prototipi funzioni** (definite altrove)
**extern** tipoJ+1 FJ+1(parametri); **extern**...
**extern** tipoN FN(parametri);
** *eventuale* definizione della **funzione main**
main(int argc, char **argv)
{
- definizione **variabili locali (auto e static)** al main
- codice del main }

** definizione della generica **funzione esterna Fy** (con y=1...K)
tipoy Fy(parametri)
{
- definizione **variabili locali (auto e static)**
- codice della funzione Fy }

** definizioni della generica **funzione statica Fx** (con x=K+1...J)
**static** tipox Fx(parametri)
{
- definizione **variabili locali (auto e static)**
- codice della funzione Fx }



## Esempio di applicazione sviluppata su più file: Uno stack

Nel seguito si codifica l'esempio della funzioni relative ad uno stack

- si definisce il file .c con le funzioni di uso dello stack
- si definisce il file .h che contiene le dichiarazioni necessarie per usare lo stack
- altri file .c (p.e., uno contenente un main) devono importare le dichiarazioni e poi possono usare lo stack).

**/* FILE STACK.h */**
```
#define SIZE 20
extern int TOP;
extern void PUSH (int x);
extern int POP (void);
```

**/* FILE STACK.C */**
```
#include <stdio.h>
#define SIZE 20

static int S [SIZE];

int TOP = 0;  /* inizializzazione */

void PUSH (int x)
{       S [TOP] = x;     TOP ++; };

int POP ()
{       TOP --;     return  S [TOP];   };
```



**/* FILE MAIN.C ===> MODULO che usa il MODULO STACK.C */**
```
#include <stdio.h>

#include "STACK.h"

void main (void     /* main */
{       if (TOP == SIZE) printf ("lo stack e' saturato");
        else  PUSH(10);
        if (TOP == SIZE) printf ("lo stack e' saturato");
        else  PUSH(100);
        if (TOP == 0) printf ("lo stack e' vuoto");
        else printf ("primo elemento estratto %d\n", POP());
        if (TOP == 0) printf ("lo stack e' vuoto");
        else printf ("secondo elemento estratto %d\n", POP());    };
```

Per creare un UNICO ESEGUIBILE ===>
bisogna prima COMPILARE INDIPENDENTEMENTE i file *prova.c* e *list.c* e poi COLLEGARE insieme i file oggetto corrispondenti *prova.obj* e *list.obj*



## CATEGORIE DI MEMORIA IN C (riassunto)

## Memoria GLOBALE e DINAMICA per i dati

1. **Memoria GLOBALE** (cioé **statica**):
   Vengono memorizzati i **DATI GLOBALI** cioè le variabili globali definite nel programma
   ⇨ classe di memoria: **extern** o **static**
   Allocazione implicita all'inizio del programma (e deallocazione al termine) a carico del compilatore
2. **Memoria DINAMICA**: ci sono due categorie di locazioni
   - **STACK**
     vengono memorizzate le variabili locali a blocchi/funzioni e i parametri delle funzioni
     ⇨ classe di memoria per le variabili: **auto**
     Allocazione implicita all'inizio del blocco/funzione per auto (e deallocazione al termine) a carico del compilatore
   - **HEAP**
     vengono memorizzate le locazioni accedute tramite puntatori
     **tempo di vita del dato riferito:** *dipendente dall'utente*
     allocazione *esplicita* tramite **malloc()**
     e analogamente deallocazione *esplicita* tramite **free()**
     **visibilità del dato riferito:** *dipendente dall'utente*

## Memoria GLOBALE per le funzioni

**CODICE GLOBALE** corrispondenti a funzioni definite nel programma
⇨ classe di memoria: **extern** o **static**

***A livello di implementazione:***
ogni entità (variabile o funzione) a qualunque categoria appartenga è memorizzata nella memoria del calcolatore, ma a seconda della categoria viene inserita nella zona di memoria gestita come:

- DATI GLOBALI ⎫
- STACK ⎬ per le variabili
- HEAP ⎭
- CODICE per le funzioni

che sono comunque gestite come parti logicamente separate

Esiste inoltre una memoria interna al microprocessore: **REGISTRI** usata per variabili e parametri register (per auto, eventuale, ottimizzazione del compilatore)

L'allocazione può avvenire secondo il seguente schema:

`