Programma svolto in Fondamenti di Informatica 1

1. Architettura dei sistemi di elaborazione.
   - Struttura generale di un calcolatore elettronico, macchina di Von Neumann.
2. Software di base per sistemi di elaborazione: il sistema operativo.
   - Il sistema operativo Window, il file system.
3. Metodi per l'analisi di un problema.
   - Algoritmi e programmi.
4. I linguaggi di programmazione e cenni alla loro evoluzione.
   - Ambienti di programmazione: editor, debugger, compilatori ed interpreti; fasi di sviluppo di un programma. Sviluppo top-down e bottom-up.
5. Il linguaggio C.
   - Alfabeto e sintassi del C, Tipi di dato primitivi in C, Tipi di dato scalari e strutturati.
   - Espressioni.
   - Dichiarazione di costanti, variabili e loro tipo.
   - Istruzioni di assegnamento e di ingresso/uscita, composte, condizionali e cicli.
   - Funzioni e procedure, Ricorsione e record di attivazione.
   - Tecniche di passaggio dei parametri, Regole di visibilità e tempo di vita.
   - Librerie standard.
   - Gestione di file binari e di testo.
   - Il preprocessore C, il linker.
   - Progetti su più file.

Astrazioni:

- sul controllo (programmazione strutturata):

  sequenza (;), blocco, **if else**, **while do**, etc.

```
    for (i=1 ; i <= n ; ++i)  p *= x;
    scanf("%f",p);
}
```

- sulle operazioni (procedure e funzioni):

```
long potenza(int x, int n) /* parametri formali */
   {   int    i;
    long   p = 1;          /* variabili locali */
    for (i=1 ; i <= n ; ++i)  p *= x;
    return p;
}
```

Chiamata:

```
long Y;
...
Y=potenza(X,10);
```

- sui dati (astrazioni di dato, tipi di dato astratti, classi e oggetti)

```
public class Counter {
    private int x;
    public void reset() { x = 0; }
    public void inc(int n)   { x = x + n; }
    public int getValue() { return x;}
}
```

```
Counter Y;
Y = new Counter();
Y.reset();
Y.inc(20);
```

FUNZIONI E PROCEDURE (richiamo)

Le astrazioni di *funzione* e *procedura* sono presenti in tutti i linguaggi di programmazione di alto livello.

Una **funzione** è un *componente software* che cattura l'astrazione matematica di funzione:

$$f : A \times B \times ... \times Q \rightarrow S$$

- molti possibili ingressi (che non vengono modificati!)
- **una sola uscita** (il risultato)

Una **procedura** è un *componente software* che cattura l'astrazione di *azione complessa* su determinati *oggetti*

- uno o più oggetti (che possono anche essere modificati)
- **possibili più uscite (event. coincidenti con gli ingressi)**

Funzioni e procedure consentono di creare delle unità di traduzione, dotate di *nome*, che *incapsulano le istruzioni che realizzano un certo servizio.*

All'atto della chiamata, ***l'esecuzione del cliente viene sospesa* e *il controllo passa al servitore*.**

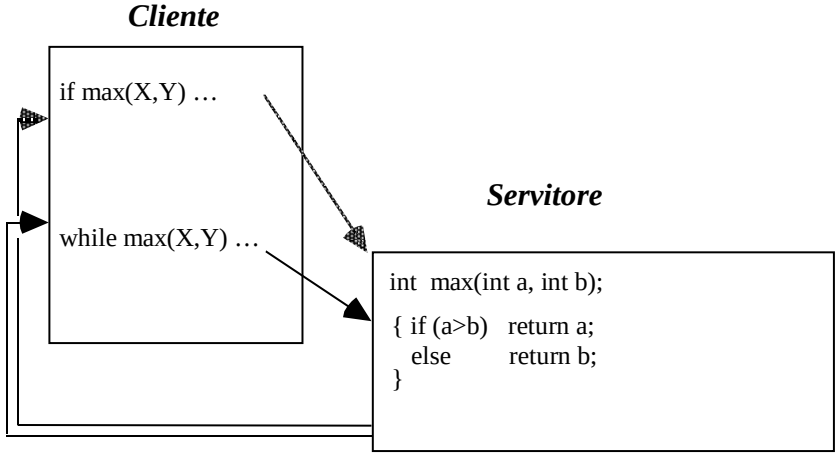
Per chiedere al servitore di svolgere un servizio occorre **chiamarlo per nome**, e *passargli le necessarie informazioni***.**

FUNZIONI E PROCEDURE

COMUNICAZIONE fra cliente e servitore

Cliente e servitore comunicano mediante:

- i ***parametri*** trasmessi dal cliente all'atto della chiamata del servitore
- l'eventuale ***valore restituito*** dal servitore al cliente



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***)

FUNZIONI E PROCEDURE: USO

Funzione:

Scrivere una funzione che calcoli la potenza N-esima di un valore dato X.

SPECIFICA

Moltiplicare X per se stesso N volte

CODIFICA

```
long potenza(int x, int n) /* parametri formali */
   {   int    i;
    long   p = 1;          /* variabili locali */
    for (i=1 ; i <= n ; ++i)  p *= x;
    return p;
}
```

ESEMPIO D'USO

```
main(){
int risultato;
risultato = potenza(4,3);      /* 4^3 = 64 */
}
```

È l'astrazione del concetto di *operatore*

Procedura:

Scrivere una procedura che stampi il maggiore tra due numeri interi.

SPECIFICA

```
void stampa_max (int a, int b);
```

<stampa il valore maggiore tra a e b>

CODIFICA

```
void stampa_max (int a, int b)        /* parametri
formali */
{   if (a>b)   printf("%d",a);
    else       printf("%d",b);
}
```

ESEMPIO D'USO

```
main(){
int X=10, Y=100;
stampa_max(X,Y);      /* parametri attuali */
}
```

È un'astrazione della nozione di *istruzione* → p**uò comparire ovunque possa comparire un'istruzione**

PASSAGGIO DEI PARAMETRI

Per "passaggio dei parametri" si intende l'associazione fra parametri attuali e parametri formali che avviene al momento della chiamata di una funzione (o procedura).

Il meccanismo descritto, adottato in C,
si chiama PASSAGGIO PER VALORE

eccettuati i vettori, dei quali si trasferisce l'indirizzo

LIMITI:

- il passaggio per valore *impedisce a priori* di scrivere procedure che abbiano come scopo *modificare* i dati passati dal chiamante

Il passaggio per riferimento NON è disponibile in C:
deve essere *simulato* tramite puntatori

E' disponibile però in C++

ESEMPIO

Data una temperatura, espressa in gradi Celsius oppure Fahrenheit, scrivere un programma che contenga **due funzioni di conversione** (Fahrenheit → Celsius, Celsius → Fahrenheit), e calcoli il corrispondente valore espresso sia in Fahrenheit sia in Celsius.

SPECIFICA DEI DUE SERVITORI

```
int fahrToCels(int fahr);
```

<restituisce il valore di fahr convertito in gradi Celsius>

```
int celsToFahr(int cels);
```

<restituisce il valore di cels convertito in gradi Fahrenheit>

SPECIFICA DEL CLIENTE

```
main() {
char scala = 'C';
int  t = 20, c, f;
 if (scala == 'F') c = fahrToCels(f = t);
 else              f = celsToFahr(c = t);
}
```

ESEMPIO / segue

```
const float F1 = 9.0, F2 = 5.0, SH = 32.0;  /*
globali */

int fahrToCels(int fahr);          /* prototipi */
int celsToFahr(int cels);

main(){
char scala = 'C';                  /* var. locali
al main */
int  t = 20, c, f;
 if (scala == 'F') c = fahrToCels(f = t);
 else              f = celsToFahr(c = t);
}

int fahrToCels(int fahr){
    return F2 / F1 * (fahr - SH);
}

int celsToFahr(int cels){
    return SH + cels * F1 / F2;
}
```


LA MACCHINA VIRTUALE DEL C: IL MODELLO A TEMPO DI ESECUZIONE

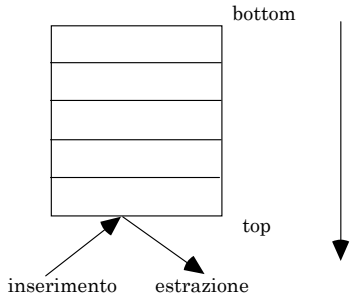
Qual è il supporto, a livello di macchina intermedia, per l'esecuzione di procedure e funzioni?

Modello a tempo di esecuzione

I parametri formali e le variabili locali:

- sono creati al momento della chiamata della funzione
- sono inizializzati con i valori dei corrispondenti parametri attuali trasmessi dal cliente
- vivono per tutto il tempo in cui la funzione è in esecuzione
- sono distrutti quando la funzione termina

Un'area appositamente allocata in uno ***stack*** (o pila) consente la loro memorizzazione in una struttura dati detta ***record di attivazione***

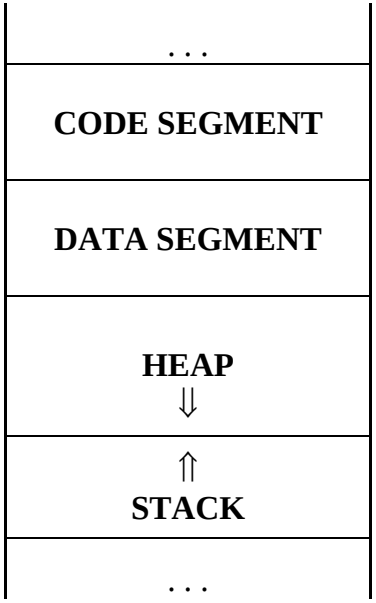


In questo modo, il main può chiamare una funzione, che ne chiama un'altra, che ne chiama un'altra …

AREE DI MEMORIA:

- area del codice — contiene il codice del programma
- area dati globali — contiene le variabili globali e statiche
- area heap — disponibile per allocazioni dinamiche
- area stack — contiene i *record di attivazione* delle funzioni (variabili locali e parametri)

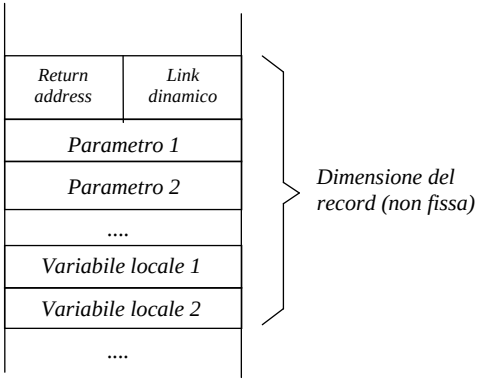
L'area stack contiene i ***record di attivazione*** delle funzioni.



RECORD DI ATTIVAZIONE

Un record di attivazione rappresenta il "mondo" della funzione, e contiene *tutto ciò che ne caratterizza l'esistenza*

- le variabili locali
- i parametri ricevuti
- l'indirizzo (del chiamante) a cui restituire il controllo alla fine dell'esecuzione della funzione (*return address*)
- un riferimento al record di attivazione del chiamante (link dinamico)
- un riferimento all'ambiente globale (link statico)



Ad ogni attivazione di funzione viene creato un nuovo record d'attivazione **specifico per quella chiamata.**

Per catturare la semantica delle chiamate annidate (una funzione che chiama un'altra funzione che...), questa area di memoria è gestita come una pila (***stack***):

Last In, First Out $\rightarrow$ *LIFO*

(l'ultimo a entrare è il primo a uscire)

RECORD DI ATTIVAZIONE (II)

La dimensione del record di attivazione:

- varia da una funzione all'altra
- ma, per una data funzione, è fissa e calcolabile a priori (la determina il compilatore, sulla base del codice sorgente della funzione).

Il record di attivazione:

- viene *creato dinamicamente* nel momento in cui la funzione viene chiamata
- rimane sullo stack per tutto il tempo in cui la funzione è in esecuzione
- viene *deallocato alla fine* quando la funzione termina (**`return`**)

Funzioni che chiamano altre funzioni danno luogo a una *sequenza* di record di attivazione

- allocati secondo l'ordine delle chiamate
- deallocati in ordine inverso

→ i record di attivazione possono essere *innestati*

RECORD DI ATTIVAZIONE (III)

Quando la funzione termina, il controllo torna al chiamante, che deve:

- riprendere la sua esecuzione dall'istruzione successiva alla chiamata della funzione (*return address*)
- trovare il suo "mondo" inalterato.

A questo scopo, quando il chiamante chiama la funzione, si inseriscono nel record di attivazione della funzione anche:

- l'**indirizzo di ritorno**, ossia l'indirizzo della prossima istruzione del chiamante che andrà eseguita quando la funzione terminerà
- il **link dinamico**, ossia un collegamento al record di attivazione del chiamante, in modo da poter ripristinare l'ambiente del chiamante quando la funzione terminerà

La sequenza dei link dinamici costituisce la cosiddetta ***catena dinamica***, che rappresenta *la storia* delle attivazioni ("chi ha chiamato chi")

Per le **funzioni**, spesso il record di attivazione prevede anche una ulteriore cella, destinata a *contenere il risultato* della funzione.

Tale risultato viene copiato dal chiamante *prima* (ovviamente!) di distruggere il record della funzione appena terminata.

Altre volte, il risultato viene restituito dalla funzione al chiamante semplicemente *lasciandolo in un registro* della CPU.

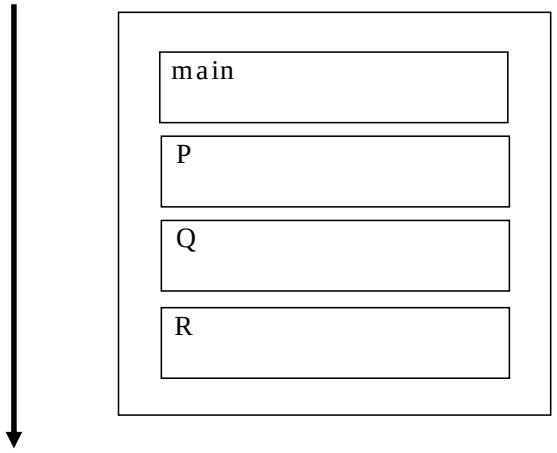
RECORD DI ATTIVAZIONE (III)

Ad esempio, sia dato il seguente programma:

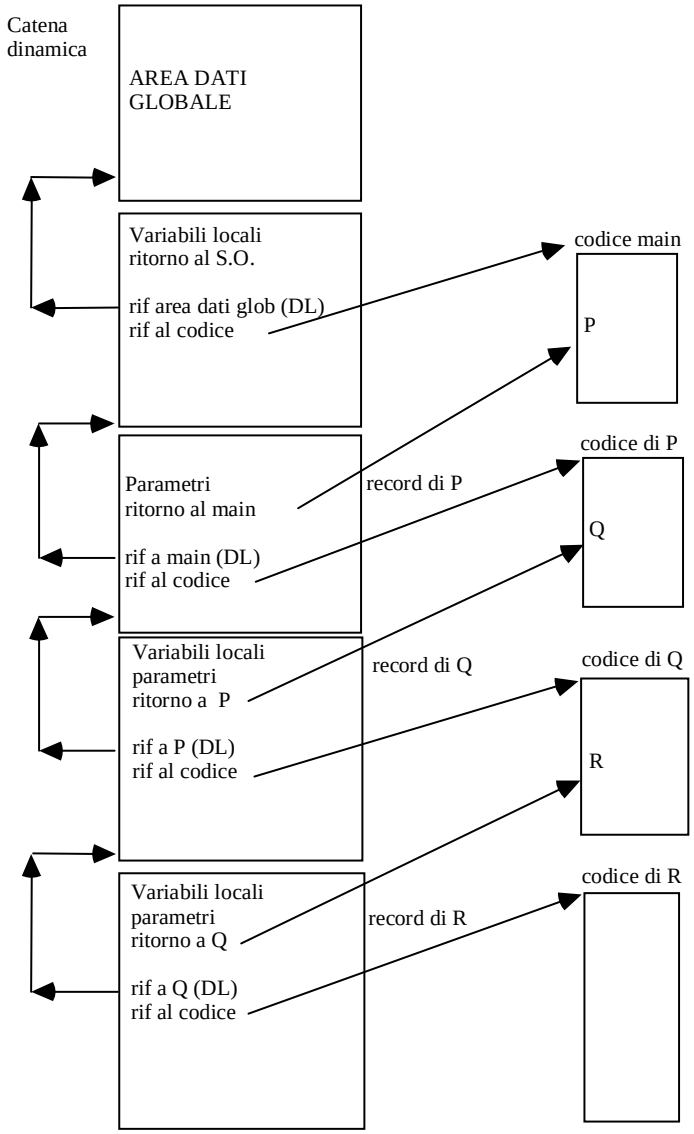
```
int x = 4;

void R(int A) {  x = A; }

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

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

main() {  P();  }
```

e supponiamo che la funzione P (attivata da main) abbia attivato Q che a sua volta ha attivato R.

Attivazioni: (S.O. $\rightarrow$) main $\rightarrow$ P $\rightarrow$ Q $\rightarrow$ R



CATENA DINAMICA:



RECORD DI ATTIVAZIONE

E PASSAGGIO DEI PARAMETRI

In C, i parametri sono passati sempre e solo *per valore*.

Il C++ consente però anche il passaggio *per riferimento*.

Parametri passati per valore

- nella cella del record di attivazione viene copiato *il valore* assunto dal parametro attuale all'atto della chiamata.

Parametri passati per riferimento (SOLO C++)

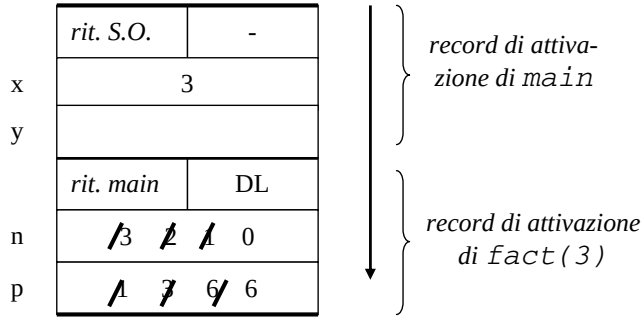
- nella cella del record di attivazione viene copiato *l'indirizzo* della variabile che costituisce il parametro attuale all'atto della chiamata.

ESEMPIO 1

```
int fact(int n) {
    int p = 1;
    while (n >0) p *= n--;
    return p;
}
```

```
main(){
  int x = 3, y;
  y = fact(x);
}
```

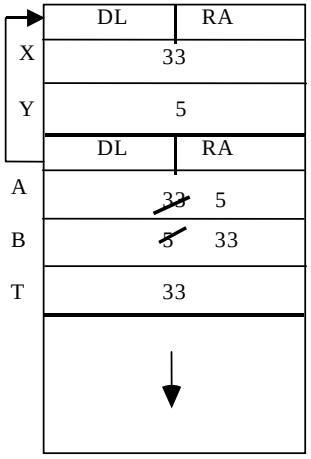
La figura illustra la situazione nello stack nel momento di massima espansione, quando la funzione sta per terminare.



Successivamente la funzione `fact` termina, il suo record viene deallocato e il risultato viene trasferito (tramite un registro macchina) nella cella di nome `y`.

ESEMPIO 2

```
main() {
 int X = 33, Y = 5;
    scambia1(X,Y);
}

void scambia1(int A, int B) {
 int T;
    if (A>B) { T=A;  A=B;  B=T; }
}
```

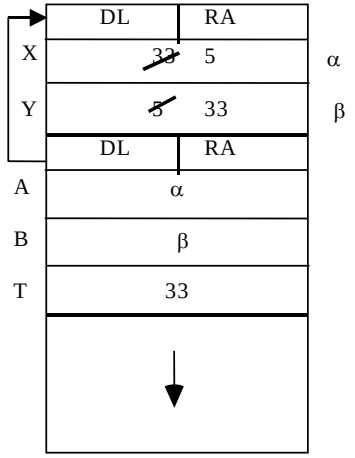


ESEMPIO 3 (passaggio per riferimento)

```
main() {
 int X = 33, Y = 5;
    scambia2(&X,&Y);
}

void scambia2(int * A, int * B) {
 int T;
    if (*A>*B) {   T=*A;  *A=*B;  *B=T; }
}
```

Quando il chiamante invoca scambia2, ora A e B non contengono più una copia di valori, ma un *riferimento* alla corrispondente variabile X o Y del chiamante

cioè **l'indirizzo (qui indicato con α, β) di tali variabili!**



Esempio 4 (passaggio di array)

```
#include <stdio.h>
void main(void);              /*  Opzionale  */
void prova(int a[ ], int b, int n);

void main(void)
{
    int c[3], d;
    c[0] = 100; c[1] = 15;
    c[2] = 20;  d = 0;
    printf("Prima: %d,%d,%d,%d\n",
           c[0],c[1],c[2],d);
    prova(c,d,3);
    printf("Dopo: %d,%d,%d,%d\n",
           c[0],c[1],c[2],d);
}

void prova(int a[ ], int b, int n)
{                 /*  a per riferimento; b,n per
valore  */
    int i;
    for (i = 1; i < n; i++) a[i] = b;
    b = a[0];
}
```

Il risultato dell'esecuzione di questo programma è:

Prima: 100,15,20,0

Dopo: 100,0,0,0 **d ?**

Dichiarazioni che alterano la visibilità/tempo di vita delle variabili:

- **auto** *automatiche (locali)*

  locali ad un blocco (blocco o funzione)

  L'entità è visibile solo all'interno nel blocco

  non visibile all'esterno

  come **variabili locali ad una procedura**

- **extern** *esterne (globali)*

  dichiarazioni riferite a variabili globali

  visibili a tutto il programma (anche in file diversi)

- **static** *statiche (globali)*

  variabili statiche interne alle funzioni

  non sono visibili all'esterno di queste

default: extern le globali

auto le locali

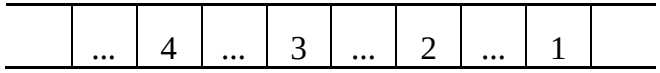
CLASSE di MEMORIZZAZIONE auto

- automatica - **default** per **variabili locali**, non si applica alle funzioni
- **visibilità locale**: la variabile è visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi
- la variabile è **temporanea**: esiste dal momento della definizione, sino all'uscita dal blocco o dalla funzione in cui è stata definita
- su **STACK**

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

```
fattoriale(int n)  /*  solo n >= 0  */
{
if (n <= 1) return 1;
else return n * fattoriale(n - 1);
}
```

```
    ... fattoriale(4) ...
```



CLASSE di MEMORIZZAZIONE extern

- esterna - **default** per **variabili globali** e **funzioni**
- **visibilità globale**: visibile ovunque, dal punto di definizione (o dichiarazione) in poi

  visibile anche **al di fuori del file** che ne contiene la definizione
- **permanente**: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine
- su **DATA SEGMENT** (**variabili** - valore iniziale di default 0) oppure
- su **CODE SEGMENT** (**funzioni**)

File "AAA.c"

```
extern        void
fun2(...);
...
int ncall = 0;
...

fun1(...)
{ncall++;
 ...
}
```

File "BBB.c"

```
extern fun1(...);
void fun2(...);
...
extern int ncall;
...

void fun2(...)
{
  ncall++;
   ...
}
```

la variabile ncall e le funzioni fun1 e fun2 sono visibili ed utilizzabili in entrambi i file

CLASSE di MEMORIZZAZIONE static

- statica - definizione globale o locale
- **visibilità**:

  globale nel caso di definizione globale: visibile ovunque, dal punto di definizione (o dichiarazione) in poi, ma **solo all'interno del file che la contiene**

  locale nel caso di definizione locale (solo variabili): visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi
- **permanente**: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine
- su **DATA SEGMENT** (**variabili** - valore iniziale di default 0) oppure
- su **CODE SEGMENT** (**funzioni**)

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;
}
```


CODE SEGMENT

- le funzioni nel segmento codice

DATA SEGMENT

- variabili extern (globali multi-file)
- variabili static (globali single-file e locali)

STACK

- variabili auto (locali - argomenti funzioni)

HEAP

- strutture dati allocate (**malloc**) e deallocate (**free**) esplicitamente dall'utente e referenziate tramite puntatori

Esiste un ulteriore classe di memorizzazione (**register**) che alloca variabili su registri macchina.