Input/Output

In C, input/output è definito dal
sistema operativo

UNIX prevede una **gestione integrata**
di **I/O** e dell'**accesso** ai file

Per I/O:

Input/Output a caratteri
Input/Output a stringhe di caratteri
Input/Output con formato
possibilità di interagire con tipi diversi

Altre librerie
accesso a strutture FILE
(fopen, fread, ... , fclose)

Azioni di base del **Sistema Operativo**
Input/Output su file
primitive di accesso
(read / write)

Modello di programma a FILTRO

Input/Output

1. Input/Output a caratteri:

int **getchar**(void); legge un carattere
o restituisce il carattere letto **convertito in int** o
EOF in caso di end-of-file o errore
int **putchar**(int c); scrive un carattere
o restituisce il carattere scritto o
EOF in caso di errore

Esempio: Programma che copia da input (la tastiera)
su output (il video):

```
#include <stdio.h>
main()
    {
    int c;
    while ((c = getchar()) != EOF)  putchar(c);
    }
```

ATTENZIONE: La funzione getchar comincia a restituire caratteri solo quando è stato battuto un carriage return (invio) e il sistema operativo li ha memorizzati

2. Input/Output a stringhe di caratteri:

char ***gets**(char *s); legge una stringa
o restituisce la stringa
  se ok, indirizzo del primo carattere o
  in caso di end-of-file o errore
  stringa nulla (**0** ossia carattere NULL)

int **puts**(char *s); scrive una stringa
o in caso di errore restituisce **EOF**

Esempio (lo stesso di prima):

```
#include <stdio.h>
main()
    {
    char s[81];
    while (gets(s))  puts(s);
    }
```

o le **stringhe di caratteri** vengono memorizzate **in array di caratteri**
o le stringhe di caratteri **terminano con** il carattere **'\0'** (**NULL** - valore decimale zero)
o la gets **sostituisce** il **new line con** il **NULL**
o la puts **aggiunge un new line** alla stringa

```
putchar('A'); putchar('B'); puts("C"); putchar('D');
ABC
D
```


3. Input/Output con formato:

Si forniscono funzioni per la lettura/srittura di dati formattati di tipo molto diverso
si noti il numero variabile dei parametri

int **printf** (char *format, expr1, expr2, ..., exprN);
  o scrive una serie di valori in base alle specifiche contenute in format
  o i valori sono i risultati delle espressioni expr1, expr2, ..., exprN
  o restituisce il numero di caratteri scritti, oppure EOF in caso di errore

int **scanf** (char *format, &var1, &var2, ..., &varN);
  o legge una serie di valori in base alle specifiche contenute in format
  o memorizza i valori nelle variabili var1, var2, ..., varN **passate per riferimento**
  o restituisce il numero di valori letti e memorizzati, oppure EOF in caso di end-of-file

Esempi:

```
int k;
scanf("%d",&k);
printf("Il quadrato di %d e' %d",k,k*k);
```

Formati più comuni

		short	long
signed int	**%d**	**%hd**	**%ld**
unsigned int	**%u** (decimale)	**%hu**	**%lu**
	%o (ottale)	**%ho**	**%lo**
	%x (esadecimale)	**%hx**	**%lx**
float	**%e**, **%f**, **%g**		
double	**%le**, **%lf**, **%lg**		
carattere singolo		**%c**	
stringa di caratteri		**%s**	
puntatori (indirizzi)		**%p**	

Per l'output dei caratteri di controllo si usano:
'\n', '\t', etc.

Per l'output del carattere '%' si usa:
%% \% non funziona!

```
    printf("%x, %o, %%",70000,70000);/*  NO!  */
1, 10560, %

    printf("%lx, %lo, %%",70000,70000);    /*  SI!  */
11170, 210560, %
```


Esempi

```
main()
  {
        float x;
        int ret, i;
        char name[50];
   printf("Inserisci un numero decimale, ");
   printf("un floating ed una stringa con meno ");
   printf("di  50 caratteri e senza bianchi");
   ret = scanf("%d%f%s", &i, &x, name);
   printf("%d valori letti %d %f %s", ret, i, x, name);
   }
```

```
main()
  {
        int a;
   printf("Dai un carattere e ottieni il valore \
decimale, ottale e hex  ");
   a = getchar();
   printf("\n%c vale %d in decimale, %o in ottale \
e %x in hex.\n",a, a, a, a);
   }
```

Chi interpreta i caratteri % nelle stringhe di formato?

Accesso ad alto livello: strutture FILE

```
#include <stdio.h>
#define  MAX 80
main  (int argc; char ** argv)
{  char * f1, * f2;
    FILE * infile, * outfile;
     int nread, b;  char c, a, buf [MAX];
...
/* prologo: apertura dei file interessati */
/* le aree puntate da infile ed outfile non sono allocate  */
   if (( infile  = fopen ( f1, "r")) == NULL) exit (1);
   if (( outfile = fopen ( f2, "w")) == NULL)
      {  fclose (infile); exit (2); }

/*  operazioni a linea  */
   while ( fgets ( buf, MAX, infile) != NULL)
        fputs (buf, outfile); /* cioè uso di linee in I/O */
/* operazioni sicure, ma difficili da trattare */

/*  operazioni a blocchi */
   while ((nread = fread (buf, 1, MAX, infile)) > 0 )
         fwrite (buf, 1, MAX, outfile );

   while
    ((nitemread = fscanf (infile, "%c %d %c ", &a, &b, &c)) > 0 )
      if (nitemread == 3) fprintf (outfile, "%c %d %c ", a, b, c);

/* epilogo: chiusura dei file interessati */
  fclose (infile);  fclose (outfile);

}
```


COSTRUTTORI DI TIPO

Sono presenti gli usuali **costruttori**
per formare strutture dati complesse:

enumerazione		*enumerazione esplicita*
array	array	*ripetizione enumerativa*
structure	record	*sequenza*
union	varianti di record	*alternativa*
puntatori	puntatori	*goto*

non ci sono i ***set*** ed i ***file***

dichiarazione di un **tipo**

formato generale:

typedef vecchiotipo costruttore nuovotipo;

definizione di una **variabile**

formato generale:

tipoelemento nomevariabile;

Una variabile può essere inizializzata alla definizione
La inizializzazione segue la **definizione** (dopo **=**) e riferisce sempre la entità in definizione

Esempio
typedef float bfl [10];
Il tipo bfl determina un array di floating da 0 a 9
bfl bfloat = {0.,0.,0.,0.,1.,0.,0.,0.,0.,0.,};

MANCANZE in DISCIPLINA

Le deroghe alla disciplina di *strutturazione* e *programmazione* in C sono introdotte

per la *programmazione* ***di sistema***

però possono portare a
programmazione *oscura* e di difficile *riusabilità*.

ancora di SISTEMA

Stretta relazione tra array e puntatori {sic}

Gli array ed i puntatori sono considerati come la stessa **notazione**
nome di un array ==
un puntatore al suo primo elemento,
➠ **possibilità di incremento/decremento sugli indirizz**i
dei puntatori

```
char v1[10],  *v2;
/* v1    è una costante e come nome equivale a &v[0]     */

/* v1 = v2; NO */
v2 = v1;

v1[0]       equivale a      *(v1)
v1[1]       equivale a      *(v1 + 1)
v1[expr]    equivale a      *(v1 + expr)
```


ARITMETICA SUGLI INDIRIZZI

ogni riferimento ad un elemento di un array è espanso come un ***puntatore*** *dereferenziato* e spiazzamento rispetto al primo elemento

```
int  arr [10],    *puntarr;

puntarr = arr;
arr [0] equivale a *arr  ===>
arr [0] equivale a *puntarr

Non solo  * arr   ma /* risic */
* (arr + 2) si riferisce il terzo elemento
(anche se non è un primitivo)
data arrdata [5], *puntdate;
```

*** (arrdata + 1)** è la seconda data
nell'array arrdata costituito da 5 elementi data
equivale a **arrdata [1]**

il Compilatore **non controlla** ma fa solo eseguire la somma dell'indice (scalato) con l'indirizzo del primo elemento
a[i] e **i[a]** sono lo stesso
===> *** (a + i)** === *** (i + a)**

Esempio:

```
main ()
{ char a[]= "0123456789";
  int i = 5;
      printf ("%c %c %c %c\n", a[i], a[5], i[a], 5[a]);
}
```

si stampa:
5 5 5 5 {SIC**!!!**}
Ah, i risultati della equivalenza di nome

Questo non vuole dire che
array e puntatori siano equivalenti!

array

area di memoria allocata totalmente
(dimensioni fissate)
costante come nome

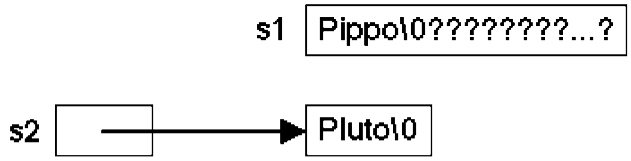
puntatori

area di memoria da allocare
(malloc o assegnamenti)
variabile con possibilità di cambiare valore

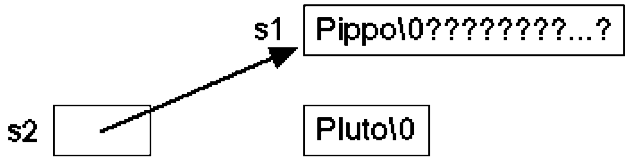
Esempio:

```
char s1[81] = "Pippo",*s2 = "Pluto";
```

s1 array di caratteri di **dimensioni fisse** (81) che può contenere una stringa di caratteri di **lunghezza variabile** (da 0 a 80 caratteri) - inizializzato a "Pippo"
s2 puntatore a carattere - inizializzato all'indirizzo del primo carattere della stringa "Pluto"



non posso scrivere s1 = s2; s1 è una costante
ma posso scrivere s2 = s1;
e se scrivo s2 = & s1; ottengo l'effetto precedente



L'area di memoria contenente la stringa "Pluto"
non è più referenziata e non è più referenziabile!

Esercizi

```
int *p1,p2[ ] = {1,2,3,4,5},k;
char *s1,s2[ ] = "12345",*s3 = "67890"

p1 = 0;   *p1 = 0;
p1 = (int *) 0;
p1 = 55;    p1 = (int *) 55;

printf("%d",p1);
printf("%d",*p1);
printf("%d",&p1);

s1 = "";
s1 = '\0';        s1 = "\0";
s1 = "abc";     s1 = &"abc";

p1 = (int *) malloc(sizeof(int)*10);
s1 = (char *) malloc(81);

for (k = 0; k < 5; k++)  p1[k] = p1[k+5] = p2[k];

free(p2);

s3 = s1;
free(s3);
for (k = 0; k < 5; k++) *s1++ = s2[k];
```


Stringhe

DATO ASTRATTO (?)

RAPPRESENTAZIONE

- stringhe array di caratteri
- L'ultimo carattere di una stringa deve essere il carattere nullo (**\0** o **NULL**).

Esempi di stringhe

```
char s1[81];
char *s2;
```

s1 è una stringa di dimensione non fissata

s2 è un puntatore a carattere: deve essere fatta una allocazione esplicita (non necessaria con la definizione precedente)

```
s2 = (char *) malloc (81);
```

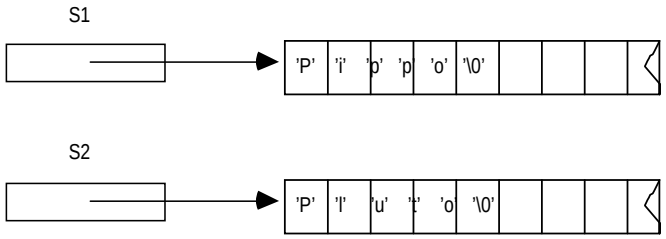
Potremmo **assegnare** da una all'altra, senza problemi

In realtà, il nome di un array è una costante

```
s1 = s2;   /* scorretta  */
s2 = s1;   /* corretta  */
```

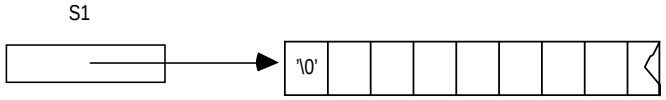
Stringhe

char s1[81];
char *s2;

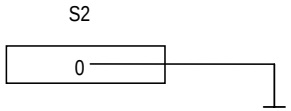


Distinguiamo tra memoria allocata
e lunghezza della stringa contenuta

Casi degni di nota



Stringa VUOTA

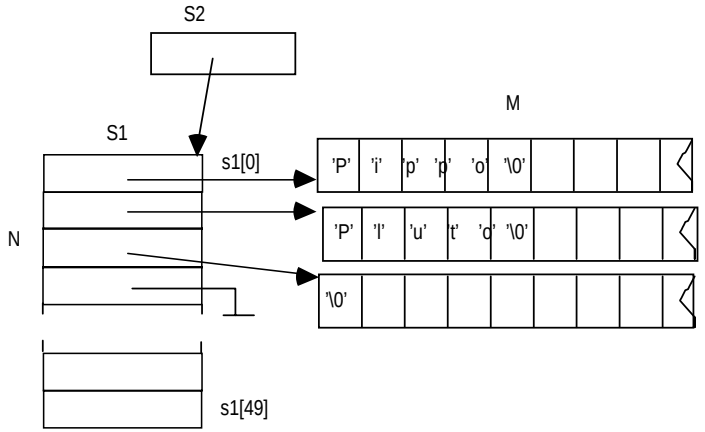


Stringa NULLA

Strutture di PUNTATORI

statiche e dinamiche

char ** s2;
char * s1 [50];



Strutture statiche
Allocazione preparata dal compilatore
NxM locazioni contigue

Strutture dinamiche
Necessità di **allocazione** della memoria
Azioni esplicite
- di allocazione
- di aggancio ad aree esistenti

STRINGHE DI CARATTERI

Stringa == Array di caratteri

```
char  string[81];    /*  max 80 caratteri  */
```

L'ultimo carattere deve essere un NULL ('\0')

```
char  text[6] = {'P','l','u','t','o','\0'};
char  text[ ] = {'P','l','u','t','o','\0'};
char  text[ ] = "Pluto";
```

text	P	l	u	t	o	\0

Dove è allocata la memoria per le stringhe costanti?

Esempio - funzione **strlen** - libreria standard C
restituisce la lunghezza di una stringa di caratteri

```
strlen(char s[ ])
  {
  int j;
  for (j = 0; s[j] != '\0'; j++) ;   /*  ... ; s[j]; ...  */
  return j;
  }
```


Esempio - funzione **strcmp** - libreria standard C
confronta due stringhe di caratteri s1 e s2
restituisce un valore:

```
< 0      se s1 <    s2
0        se s1 ==   s2
> 0      se s1 >    s2
```

```
strcmp(unsigned char s1[ ],unsigned char s2[ ])
  {
  int j = 0;
  while (s1[j] && s2[j] && s1[j] == s2[j])  j++;
  return (s1[j] − s2[j]);
  }
```

Esempio - funzione **strcpy** - libreria standard C
copia una stringa (sorgente) in un'altra (destinazione)
restituisce l'indirizzo della stringa destinazione

```
char *strcpy(char dest[ ],char src[ ])
  {
  int j = 0;
  do
    dest[j] = src[j];
  while (src[j++])
  return dest;       /*  return &dest[0]  */
  }
```

PUNTATORI e **ARRAY**

STRETTA RELAZIONE TRA PUNTATORI E ARRAY
Array e puntatori sono (quasi) **equivalenti** come **nome**
nome di un array == puntatore al primo elemento

```
int a[10],*p;
```

ARITMETICA DEI PUNTATORI

Il compilatore C esegue sempre la conversione

a[k] ➠ *(a+k)

Si noti che, applicando l'operatore & ad entrambi i termini si ottiene:

&a[k] == a+k

a+k rappresenta la sequenza di interi che inizia al k-esimo posto

Il compilatore **non esegue alcun controllo** e genera sempre il codice per eseguire la somma dell'indirizzo del primo elemento dell'array con l'indice scalato

```
long int a[ ] = {100,200,300,400,500};
```

a[-1]	?	:A-4
a[0]	100	:A
a[1]	200	:A+4
a[2]	300	:A+8
a[3]	400	:A+12
a[4]	500	:A+16
a[5]	?	:A+20

UN PUNTATORE È UNA VARIABILE

```
p = a;      /*  *p == a[0]  */
p++;        /*  *p == a[1]  */
```

IL NOME DI UN ARRAY È UNA COSTANTE

```
a = p;      /*  NO!  */
a++;        /*  NO!  */
```

ARGOMENTO FORMALE SEMPRE VARIABILE

```
fun(..., int a[ ], ...) { ... }
fun(..., int *a, ...) { ... }        /*  Equivalenti  */
```

Esempio - funzione **strlen**
restituisce la lunghezza di una stringa di caratteri

```
strlen(char *s)
  {
  int j;
  for (j = 0; *s++; j++) ;
  return j;
  }
```

Esempio - funzione **strcmp**
confronta due stringhe di caratteri s1 e s2
restituisce un valore:

```
< 0     se s1 <    s2
0       se s1 ==   s2
> 0     se s1 >    s2
```

```
strcmp (unsigned char *s1,unsigned char *s2)
  {
  while (*s1 && *s2 && *s1 == *s2)  s1++, s2++;
  return (*s1 – *s2);
  }
```

Esempio - funzione **strcpy**
copia una stringa (sorgente) in un'altra (destinazione)
restituisce l'indirizzo della stringa destinazione

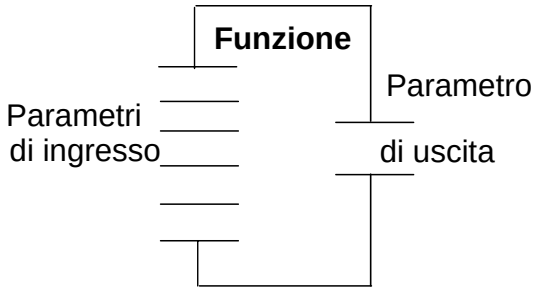
```
char * strcpy (char *dest,char *src)
  {
  char *p = dest;

  while (*dest++ = *src++)  ;
  return p;
  }
```


PUNTATORI vs. ARRAY

```
/*   array di puntatori a caratteri e puntatori di puntatori
     sono per alcuni aspetti equivalenti  */
#define MAX 100
#define NULL (char *) 0

void proc (n, arg)
/* procedura di stampa vettore o lista stringhe */
 int n; char ** arg;
{       int j;
     for (j=0; j<n; j=j+1)  /* array di stringhe */
     printf("Stringa %d vale %s\n", j, arg[j]);

     j=0;
     while (*arg)              /* doppia lista  */
     {  printf("Stringa %d vale %s\n", j, *arg);
        j=j+1;     arg=arg+1;
}    }

main ()
{ int i,n;   char *s[MAX];
   printf("Dammi n\n");  scanf("%d", &n);
   for (i=0; i<n; i=i+1)
        {  s[i] = (char *) malloc (81);
           printf("Stringa %d ", i); scanf("%s", s[i]);
        }
   s[i] = NULL;   /* stringa nulla */
   proc(n, s);
}
```

Funzioni che ritornano funzioni:
FUNZIONALI

```
int selecta (int a, int b)    { return a + b; }
int selectb (int a, int b)    { return a - b;  }
int selectc (int a, int b)    { return a * b;  }

funptr * select2 (int a)
{ switch (a)
{ case 1: return selecta; break;
  case 2: return selectb; break;
  case 3: return selectc; break;
  default: return selecta;
}
}
```



Funzionali sono funzioni i cui parametri di ingresso o di uscita sono ancora funzioni

Variabili

VISIBILITÀ (MODULI E BLOCCHI)
possibilità di riferire la entità
TEMPO di VITA (BLOCCHI)
durata della entità all'interno del programma

i MODULI statici sono FILE
i BLOCCHI dinamici sono funzioni o blocchi

VISIBILITÀ

auto *automatiche (locali)*
locali ad un blocco (blocco o funzione)
La 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

static *statiche (globali)*
variabili statiche interne alle funzioni o moduli
sono visibili dove sono stati definite
ma non sono visibili all'esterno

default: **extern** per le variabili globali
auto per le variabili locali

MEMORIA STATICA e DINAMICA

Memoria **statica** ==>
variabili globali definite nel programma principale

Memoria **dinamica** ==>
variabili locali alle funzioni e con tempo di vita
pari alla esecuzione delle funzioni

Memoria **dinamica** ==>
variabili senza nome accedute attraverso puntatori

DATI STACK HEAP separati

Tramite funzioni di libreria (**malloc**)
analogamente deallocazione (**free**).

```
puntatore = (datatype*) malloc (sizeof (datatype));
```

La **malloc alloca** un certo **numero di byte,** fornendo **un puntatore** a questi
Il tipo è **puntatore a carattere**

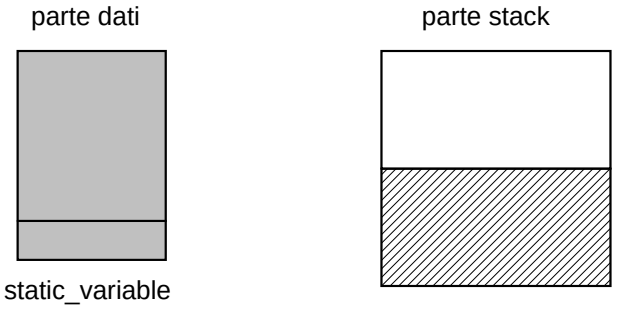
```
int *ptr;
ptr = (int *) malloc (sizeof (int));
* ptr = 55;
```


Esempio automatiche e statiche

```
static_demo ();
main()
{  int i;
     for( i= 0; i < 10; ++i)        static_demo();
 }

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

variable automatica
visibile e **presente solo** durante la invocazione
sempre a 0 ad ogni invocazione

static_variable
viene allocata come globale (una volta sola) e
visibile solo durante la invocazione
qui è incrementata ad ogni chiamata



TEMPI DI VITA delle VARIABILI

allocazione dei dati
- **automatici**: allocazione **locale**
  tempo di vita la procedura di definizione
  I dati sono locali al blocco di dichiarazione
  Allocazione e deallocazione al termine del blocco o procedura
Politica realizzata tramite ***stack***

- **statici/extern**
  Allocazione **Globale**
  tempo di vita pari al programma
  Una variabile statica interna ad una funzione permane oltre la singola invocazione della procedura.
  Ogni invocazione della stessa procedura utilizza il valore precedente della variabile.
Politica di allocazione attraverso ***dati statici***

- **dinamici**: allocazione **dinamica** dei dati riferiti attraverso puntatori
  tempo di vita dipendente dall'utente ma non legato ad un puntatore specifico, ma ad azioni di deallocazione
  l'area di memoria deve essere esplicitamente allocata/deallocata, usando le funzioni del sistema operativo (**malloc/free**)
Politica realizzata attraverso una gestione di **memoria** ad ***heap***

Classi di memorizzazione

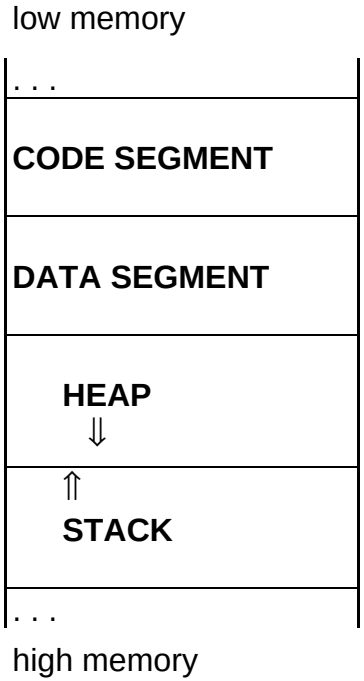
TEMPI di VITA
VISIBILITÀ

Ogni entità (variabile o funzione) ha:

- un **NOME** che la identifica (in modo univoco ?)
- un **TIPO** che identifica l'insieme dei valori ammessi e la rappresentazione interna della variabile o del risultato della funzione
- un **VALORE** tra quelli ammessi dal tipo
- un **INDIRIZZO** relativo al primo byte del blocco di memoria che contiene il valore della variabile o il codice della funzione
- una **CLASSE di MEMORIZZAZIONE** che indica il tipo di area di memoria in cui la variabile o la funzione viene memorizzata

dati	funzioni
DATA SEGMENT STACK HEAP REGISTRI	CODE SEGMENT

Un esempio di
Struttura della memoria a RUN-TIME



Come scoprire eventuali collisioni tra STACK e HEAP?

- il S.O. del Macintosh chiama 60 volte al secondo lo "stack sniffer"
- il Turbo C++ ha un opzione in compilazione "Test Stack Overflow"

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 (**valore iniziale di default ?)

```
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) ...

	...	4	...	3	...	2	...	1	

CLASSE di MEMORIZZAZIONE register

- o come le auto
- o su **REGISTRO MACCHINA**

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

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

Cosa guadagno in quest'ultimo caso?

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
- se dati, inizializzati a 0
- su **CODE SEGMENT** (**funzioni**) oppure
- su **DATA SEGMENT** (**variabili** - valore iniziale di default 0)

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**)
- se dati, inizializzati a 0

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)

REGISTRI

- variabili register (locali - argomenti funzioni)
  --- non tutti i tipi di variabili ---

HEAP

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

VISIBILITÀ delle Entità: Dichiarazioni, Definizioni

Possibilità di spezzare l'applicazione su **più file** messi insieme al **collegamento**

La fase di **compilazione** ha necessità di informazioni

La **dichiarazione** specifica la entità ===> non implica allocazione di spazio

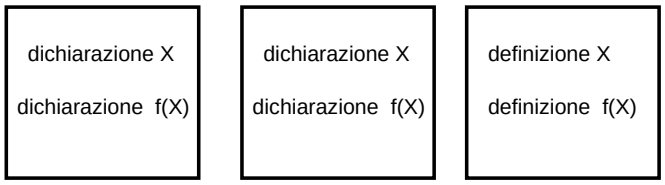
La dichiarazione serve a specificare al compilatore la struttura di variabili o funzioni allocate **in altri moduli**

La **definizione** specifica le proprietà ==> allocazione

La definizione specifica una entità da allocare in un **modulo**

Più dichiarazioni della stessa entità (in file diversi) ma **una ed una sola definizione**.

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



ESEMPIO:

file1.c

```
extern  int x;

extern float  f
        (char  c);

/*dichiarazioni
==> IMPORT */

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

file2.c

```
extern  int x;

extern float  f
        (char  c);

/*dichiarazioni
==> IMPORT */

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

file3.c

```
 int x = 10;

float  f  (char  c);
{  <var  locali  e
     codice di f >
}
/*definizioni
==> EXPORT
*/
```

COMPILAZIONE

===> INDIPENDENTE

bisogna **compilare** file1, file2 e file3

LINKING

===> RISOLVE I RIFERIMENTI ESTERNI

bisogna fare il **linking** di file1, file2 e file3 **insieme**

POSSIBILITÀ DI SVILUPPARE UN PROGRAMMA SU PIÙ FILE: VISIBILITÀ E PROTEZIONE

VARIABILI GLOBALI

static

==> entità allocate una volta per tutte, all'inizio del programma,
visibili solo nel file di definizione.

Una variabile/funzione statica definita a livello di file non è visibile al di fuori del file stesso.

Ruolo simile a quello di una **variabile/funzione *protetta e non visibile* di un modulo**

extern

==> entità dichiarate all'interno di un file per riferire entità definite in altri file
o
esportate perché possano essere riferite da altri

La clausola extern quindi è usata
sia da chi la importa
sia da chi le esporta, seppure con semantica diversa

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

Le dichiarazioni di extern sono simili alle variabili/funzioni **importate** da un modulo/unità
Le definizioni di entità extern corrispondono agli **export**.

chi esporta la entità, la definisce
chi importa la entità, la dichiara
Le dichiarazioni in altri file servono per collegarsi alle stesse variabili/funzioni e consentire controlli al compilatore

METODOLOGIA DI USO

le *definizioni* non usano extern ed usano il default: non compare la clausola esplicitamente

le *dichiarazioni* riportano la classe *extern*

Si noti che l'utente non conosce i file di importazione