LINGUAGGIO C

1972 progetto e sviluppo: **D.M. Ritchie**
dove: **AT&T Bell Laboratories**
scopo: rimpiazzare il linguaggio Assembler

C come un linguaggio di sistema

sistema operativo **UNIX**
multiutente
concorrente
nucleo centrale e
gran parte dei programmi di utilità in C

1978 **definizione** precisa **del linguaggio**:
B.W. Kernigham e D.M. Ritchie

1983 **C ANSI**
American **N**ational **S**tandards **I**nstitute

C++
Objective C

Linguaggio di programmazione

IMPERATIVO
STRUTTURATO A BLOCCHI
SEQUENZIALE
DI SISTEMA

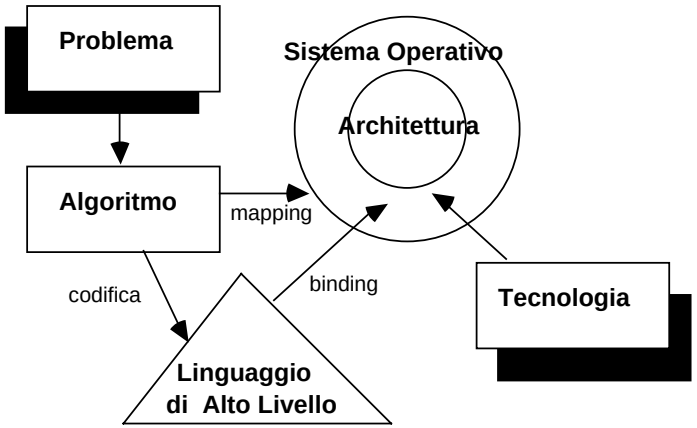
adatto per il progetto di

- software di base
- sistemi operativi
- compilatori
- DBMS
- ...

Caratteristiche del linguaggio:

- Flessibilità ➠ applicazioni di tutti i tipi
- Portabilità ➠ esempio di UNIX
  - Ampia libreria standard di funzioni
- Sinteticità ➠ attenzione alla poca leggibilità
- Semplicità ➠ pochi concetti elementari
  - dichiarazione / definizione
  - statement / blocco
  - espressione
- Efficienza

Progetto di un programma



Algoritmo
CODIFICA ==>
- in un linguaggio opportuno di alto livello

MAPPING
- decisioni di allocazione per l'architettura scelta
  Processore Intel 486, Pentium o RISC
  quanta e quale memoria associata all'heap

BINDING
- decisioni per ogni entità del programma
  le entità logiche vengono associate alle risorse fisiche
  la variabile ptr viene associata ad una area di memoria

Passi di sviluppo di un programma

PRODUZIONE del **SORGENTE**	Editor
COMPILAZIONE	Compilatore
COLLEGAMENTO	Linker
CARICAMENTO	Loader
MONITORING	Debugger

Ogni ambiente di programmazione
definisce i propri strumenti

UNIX **cc** per compilare e collegare

VAX/VMS **c** per compilare
link per collegare

Turbo C ambiente grafico integrato (project)
+
tcc per compilare (con opzioni)
tlink per collegare

Caratteristiche del **C** come **linguaggio di sistema**

- visibilità della **rappresentazione** delle variabili

  funzione **sizeof**
  per ogni variabile o tipo fornisca la dimensione

- possibilità di agire sugli **indirizzi**

  puntatori come indirizzi

  allocazione diretta in punti specifici

  aritmetica sugli indirizzi

- operatori a **basso livello**

  operatori **bit a bit**

Per le altre funzioni ci si appoggia al
sistema operativo **UNIX**:
supporto per

FILE system
Processi concorrenti e sincronizzazione
...

ANALISI ELEMENTI DI BASE
Costanti

Numeri interi

	2 byte	4 byte	
base decimale	12	70000, 12L	
base ottale	014	0210560	*inizio 0*
base esadecimale	0xFF	0x11170	*inizio 0x*

Numeri reali

24.0 2.4E1 240.0E-1

Caratteri e stringhe di caratteri

'a' 'A'
"a" "aaa" "" (stringa vuota)

Caratteri speciali:

newline	**\n**	tab	**\t**
backspace	**\b**	form feed	**\f**
carriage return	**\r**		
codifica ottale	**\ooo** con **o** cifra ottale 0-7		

\041 è la codifica del carattere !
\' \\ \" \0 (carattere nullo)

Esempi:
printf("Prima riga\nSeconda riga\n");
printf("\\\"/"); ➠ \"/

Chi interpreta i caratteri speciali?

TIPI PRIMITIVI

CARATTERI

char	caratteri ASCII
unsigned char	set di caratteri esteso

INTERI

char	-128..128
unsigned char	0..255
short int	-32768..32767
unsigned short int	0..65535
long int	-2147483648..2147483647
unsigned long int	0..4294967295

int di **16** o di **32** bit

Turbo C++

int	➠	short int
unsigned int	➠	unsigned short int

Vax C

int	➠	long int
unsigned int	➠	unsigned long int

REALI

float	32 bit
double	64 bit

BOOLEANI **non sono previsti**

il valore 0 (zero) indica **FALSO**

ogni valore diverso da 0 indica **VERO**

vengono considerati falsi:

0	'\0'	0.0	5-5

vengono considerati veri:

5	'A'	2.35	3*2

Per definire le due costanti booleane:

```
#define FALSE   0
#define TRUE    1
```

VOID

void	insieme vuoto

`void fun(...)` funzione che non restituisce alcun valore

VARIABILI

(dette in C, OGGETTI)

* **dichiarazioni**

specifica delle proprietà di una entità
solo template per verifiche di tipo

* **definizioni**

si specificano le proprietà di un'entità
con decisione di allocazione di spazio in memoria
ed eventuale inizializzazione

Tipi e variabili

Dichiarazione di un tipo

typedef tipoEsistente nomeTipo;

I tipi sono sinonimi in C

Esempi:

```
typedef  int  tipo1;
typedef  int  tipo2;
```

Definizione di una variabile

tipoEsistente nomeVariabile;

Esempi:

```
int  i, j, k;  char c = 't';
tipo1 v1;
tipo2 v2;
```

v1 e v2 sono interi

Strutturazione
del **controllo** e dei **dati**

CONTROLLO

STATEMENT C

semplici	**strutturati**
assegnamento	sequenza
break/continue	blocco
goto	if
return	switch
chiamate a funzione	for
	while

Realizzazione della uscita anomala
E l'eccezione?

DATI C

semplici		strutturati	
primitivi	**derivati**	**costruttori**	**pointer**
char	enum	array	
int		struct	
float		union	
double			

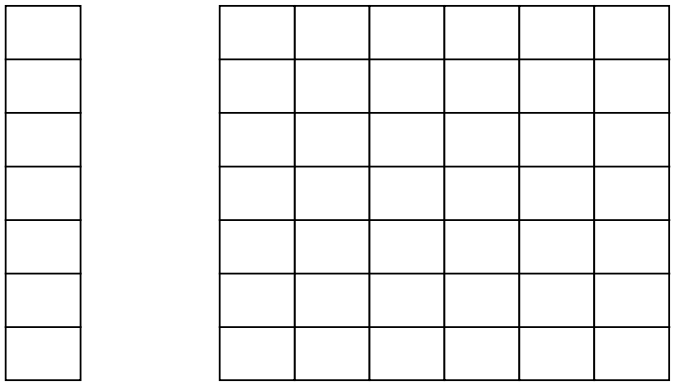
Non ci sono:
BOOLEANI
SET
FILE

vedi **UNIX** che fornisce i **file**

Differenze tra
Strutture STATICHE e
Strutture DINAMICHE

Strutture STATICHE

- *Array*
- *Matrici*



- **dimensioni** *limitate e predefinite*

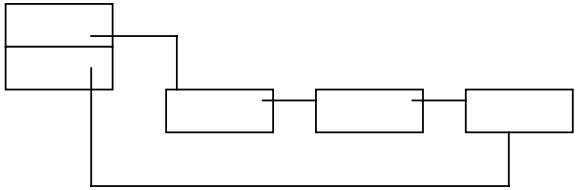
- **occupazione di memoria** *predeterminata*

- **operazioni di accesso** *facili*

- **operazioni di inserimento/estrazione**
  onerose (modifiche globali)

Strutture DINAMICHE

- *Lista*
- *Coda*
- *Stack*



- **dimensione** *illimitata*

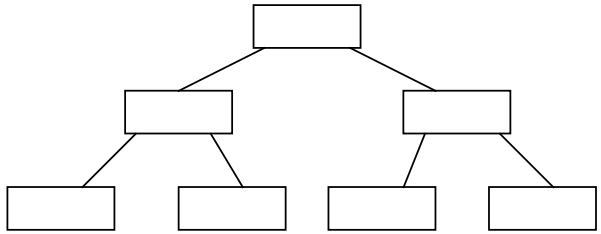
- **occupazione di memoria** *secondo necessità*
  (by need)

- **operazioni di accesso**
  sequenziali

- **operazioni di inserimento/estrazione**
  facili (modifiche locali)

Strutture DINAMICHE

Alberi
- binari
- n-ari



- **dimensione** *illimitata*

- **occupazione di memoria** *secondo necessità*
  (by need)

- **operazioni di accesso**
  proporzionali log Numeronodi

- **operazioni di inserimento/estrazione**
  facili (modifiche locali)

Differenze tra
Strutture ORDINATE e
Strutture NON ORDINATE

In caso di produzione di ***strutture dati***

===>

creare la struttura e
mantenerla nell'ordine specificato dall'uscita

Il **costo di ordinare** un *numero elevato*
di elementi può

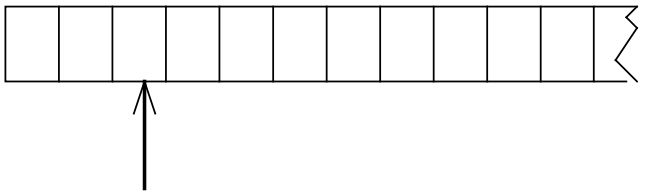
- essere elevato e
- presentare dei problemi di memoria

Il **costo di ricerca in una strututra ordinata**
con un *numero elevato* di elementi può essere molto limitato rispetto alla ricerca necessaria per strutture non ordinate

Strutture DATI PERSISTENTI

Uso di dati che vengono mantenuti tra le
diverse esecuzioni ==> **PERSISTENZA**

Il *tipo di dato astratto* **File (in UNIX)**



- **dimensioni** *illimitate*

- **occupazione di memoria secondaria** ***solo in caso di necessità***
  (by need)

- **operazioni di accesso**
  sequenziali (operazioni UNIX standard)

- **operazioni di inserimento/estrazione**
  sequenziali (modifiche globali)

Linguaggio C

- ***Limiti su Operazioni I/O:***
  Il Sistema Operativo fornisce i file e le operazioni

- ***Binding Statico:***
  Risoluzione dei nomi
  durante la compilazione o linking

- ***Passaggio Funzioni come parametri***
  Prototipo (ANSI C)

- ***Programmazione di sistema***
  Tipaggio non stretto
  Possibilità di descrivere l'architettura

LINGUAGGIO C++

- ***Oggetti in C++***
  Astrazioni dato (non protette)
  Ereditarietà (genitore multiplo)
  Polimorfismo
  Dinamicità

Espressioni

Linguaggio C
➠ **Basato su espressioni**

Espressioni
➠ **insiemi di operandi ed operatori**

OPERATORE DI ASSEGNAMENTO

var = espressione

```
j = 0;
k = j + 1;
```

Anche l'assegnamento **produce un valore**

```
i = j = k = 0;
i = 5 + k = 6;       /*  NO!  */
```

OPERATORI ARITMETICI

+ **–** * **/** **%** (modulo)

OPERATORI RELAZIONALI

== **!=** **<** **<=** **>** **>=**

OPERATORI LOGICI

&& (and) **||** (or) **!** (not)

operatori **AND** e **OR** a **CORTOCIRCUITO**
secondo operando valutato solo se necessario
verificheremo in seguito cosa produce il compilatore

```
n != 0 && k/n < 100
```

OPERATORI DI INCREMENTO/DECREMENTO

var++ utilizzo e quindi incremento
++var incremento e quindi utilizzo
var–– utilizzo e quindi decremento
––var decremento e quindi utilizzo

```
int i, j, k = 5;
i = ++k;                /*  i = 6, k = 6  */
j = i + k++;            /*  j = 12, i = 6, k = 7  */
j = ++k + k++;          /*  in cerca di guai!  */
```

CONCATENAZIONE DI ESPRESSIONI

, espr1, espr2, espr3, ..., esprN

- le espressioni vengono valutate nell'ordine di apparizione
- viene restituito il risultato dell'ultima espressione
- i risultati di tutte le altre espressioni sono scartati

Esempio - concatenazione di assegnamenti:

```
... i = 1, j = 0 ...
```

il risultato dell'espressione è 0

Espressioni (ancora)

OPERATORI SU BITS

<<	shift a sinistra	k<<4 shift a sinistra di 4 bit equivale a k∗16
>>	shift a destra	k>>4 shift a destra di 4 bit equivale a k/16
&	and bit a bit	
|	or inclusivo bit a bit	
^	or esclusivo bit a bit	
~	complemento a 1	

Scrivere le funzioni **odd**(n) ed **even**(n)

Lo shift a destra è ambiguo per valori negativi (!)
e viene lasciato alla implementazione

OPERATORE CONDIZIONALE

?: condizione ? parteVera : parteFalse

- la **parteVera** viene valutata solo se la condizione è verificata (valore diverso da 0)
- la **parteFalse** viene valutata solo se la condizione **non** è verificata (valore uguale a zero)

```
x = (y != 0 ? 1/y : INFINITY);
k = a < b ? a : b;
ptr = area < 0xF0000 ?
      area + (area >> 4)<<4) : 0xF0000;
```

ASSEGNAMENTO - FORMA ***ABBREVIATA***

v θ= e è **quasi** equivalente a v = v θ (e)

dove θ è uno dei seguenti operatori:

+ – * / % >> << & ^ |

```
k += j          /*  equivale a k = k + j  */
k *= a + b      /*  equivale a k = k * (a + b)  */
```

Le due forme sono **quasi equivalenti** perchè
in **v θ= e**, v viene valutato una sola volta, mentre
in **v = v θ (e)**, v viene valutato due volte
Se la valutazione di v non genera **effetti collaterali** (side effect) le due forme sono del tutto equivalenti,
in caso contrario le due forme non sono equivalenti

Ad esempio:

```
a [i++] *= n;
```

non è equivalente a

```
a [i++] = a [i++] * n;          /*  in cerca di guai!  */
```

bensì a

```
a [i] = a [i] * n; i++;
```


PRECEDENZA TRA GLI OPERATORI

Operatori	*Simboli*	*Associatività*
Chiamata a procedura Selezioni	() [ ] -> .	da sinistra a destra
Unari	! ~ + – ++ –– & * (type) sizeof	da destra a sinistra
Moltiplicativi	* / %	da sinistra a destra
Additivi	+ –	da sinistra a destra
Shift	<< >>	da sinistra a destra
Relazionali	< <= > >=	da sinistra a destra
Uguaglianza/Dis.	== !=	da sinistra a destra
AND bit a bit	&	da sinistra a destra
OR esclusivo bit a bit	^	da sinistra a destra
OR inclusivo bit a bit	|	da sinistra a destra
AND logico	&&	da sinistra a destra
OR logico	||	da sinistra a destra
Condizione	?:	da destra a sinistra
Assegnamenti	= += -= *= /= %= &= ^= |= <<= >>=	da destra a sinistra
Concatenazione	,	da sinistra a destra

ATTENZIONE

La **precedenza degli operatori** è una regola **sintattica**
a + b * c equivale **sempre** a a + (b * c)
la **associatività** è ancora **sintattica**
a + b + c equivale **sempre** a (a + b) + c

La **precedenza nella valutazione** degli operandi è una regola **semantica**
in C **non è definita**

Le espressioni:

a + b + c (a + b) + c

possono essere valutate calcolando
prima il primo o prima il secondo operando

L'uso delle parentesi è irrilevante - per forzare
un particolare ordine di valutazione è necessario utilizzare un'esplicita **variabile temporanea**!

```
x = f() + g();
```

se f o g alterano una variabile globale (side effect),
il valore di x può dipendere dall'ordine di valutazione

```
a[i] = i++;          /*  NO!  */
```


Analogamente **non è specificato l'ordine in cui vengono valutati gli argomenti delle funzioni**

```
printf("%d %d\n", ++n, potenza(2,n));    /*  NO!  */
```

scrivere:

```
n++;
printf("%d %d\n", n, potenza(2,n));
```

Cosa si ottiene da:

```
int var = 5;
printf("%d,%d,%d\n",var,++var+var++,var);
```

?

7,12,5 Turbo C e Vax C (senza optimizer)
7,13,5 MPW C e Vax C (con optimizer)
7,13,7 Think C

!

Esercizi

i = (j = k + 1)

i = (k = j + 12) - (j = k + 56 - j)

(k & 1)

i >>= 1 e j <<= 1

1 + ~ i (**~** è il complemento a 1 bit a bit)

~ (~ i | ~ j) vs. i & j

! (! a || ! b) vs. a && b

a && b vs. a ? b : 0

a || b vs. a ? 1 : b

! a vs. a ? 0 : 1

(x & MASK) == 0 vs. x & MASK == 0

Istruzioni e strutture di controllo

ISTRUZIONE qualsiasi espressione seguita da un punto e virgola è un'istruzione

```
x = 0; y = 1;       /*  2 istruzioni  */
x = 0, y = 1;       /*  1 sola istruzione  */
k++;
printf(...);

;  /* statement nullo */
```

BLOCCO (ISTRUZIONE COMPOSTA)

```
{   Dichiarazioni e Definizioni
    Istruzioni
}
```

ALTERNATIVA - if ... else

```
if (espressione)        istruzione1;
else                    istruzione2;
```

ALTERNATIVA MULTIPLA - switch

```
switch (espressioneIntera)
{   case costante1: blocco1; break;
    case costante2: blocco2; break;
    ...
    case costanteN: bloccoN; break;
    default: bloccoDiDefault; [break;]
}
```

ITERAZIONE - while, do, for

Ripetizioni solo non enumerative:

while (espressione)
istruzione;

do
istruzione
while (espressione);

for (espressione1; espressione2; espressione3)
istruzione;

```
for (i = 1;  i <= n;  i++)  printf("%d ",i);
for (;;)  { ... }
```

TRASFERIMENTI DI CONTROLLO

break e **continue**
uscita immediata dal **ciclo** o **switch** in cui è racchiusa
continue provoca il salto all'inizio della successiva iterazione del **ciclo** in cui è racchiusa

goto label

Usare il **goto** solo nei rari casi in cui il suo uso rende più leggibile il codice - ad esempio:

- per uscire dall'interno di strutture molto nidificate
- per convergere in caso di errore, in un unico punto da punti diversi del programma

Problemi se si entra in flusso innestato

FUNZIONI

- Tutte e solo **funzioni ricorsive** che ritornano
  - un valore (il tipo di default è **int**)
  - **void**

- **NON** definite all'interno di altre funzioni
  (una localmente ad un'altra)

- Il valore di ritorno
  non può essere un array o una function, ma
  può essere l'indirizzo di un array o di una function

- Una funzione **termina** quando viene eseguita
  - un'istruzione **return**
  - l'ultima istruzione del corpo della funzione

Ritorno al chiamante

return [espressione];

- L'espressione viene valutata e il suo valore viene restituito al chiamante
- Il tipo dell'espressione deve coincidere con il tipo della funzione
- Se la funzione è di tipo **void** solo **return**
- Nel corpo della funzione possono esserci più return

Definizione di una funzione

[tipoRisultato] nomeFunzione **(**[lista parametri]**)**
{
  [Dichiarazioni e definizioni]
  [Istruzioni]
}

Esempi:

```
max(int a, int b)
{  if (a > b) return a;
   else return b;
}

max(a, b)
int a, b;{  return (a > b) ? a : b;  }

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

funzioneAncoraDaFare(...)  {  }
```


Dichiarazione di una funzione - PROTOTIPO

[tipoRisultato] nomeFunzione **(**[lista parametri]**);**

È sempre necessario dichiarare le funzioni se il loro uso precede la loro definizione

È buona norma **utilizzare sempre i prototipi**

Il **prototipo di una funzione** è una dichiarazione che permette di controllare il tipo degli argomenti:

[tipoRisultato] nomeFunzione **(**p1Type p1Name, ...**);**
[tipoRisultato] nomeFunzione **(void);**

Esempi:

```
fun(void);            /* Funzione senza argomenti */
void proc1(int x[ ]);
max(int a, int b);
main(void);   /*  Prototipo del main  */
```

Invocazione di una funzione
... nomeFunzione **(**[lista parametri attuali]**)** ...

```
if (max(a, b) > max(c, d))  ...
printf("Massimo = %d",max(x,y));
```

Passaggio dei Parametri

- Di norma, **passaggio per valore** viene passato il valore corrente del parametro -- inefficiente per strutture dati complesse --
- Array e funzioni: **passaggio per riferimento** viene passato l'indirizzo iniziale dell'array o della funzione
- È possibile **forzare il passaggio per riferimento**, passando come parametro un indirizzo, ad esempio il puntatore a una struttura dati complessa

Se si utilizzano i prototipi, il COMPILATORE:

- **controlla la congruenza** tra il tipo del parametro attuale passato alla funzione e il tipo del parametro formale dichiarato nel prototipo
- quando necessario, **converte automaticamente** il tipo del parametro attuale passato alla funzione (sempre che i tipi siano congruenti tra loro)

PROTOTIPI ==> **MAGGIORE CONTROLLO**

Esempi di passaggi per valore e per riferimento

```
void swap(int a, int b)    /*  a, b passati per valore  */
  {  int t;
     t = a;  a = b;  b = t;
  }
```

chiamando swap(x, y)
NON SI HA ALCUN EFFETTO!
La versione corretta usa i riferimenti

```
void swap(int *a, int *b)
/* a, b ancora passati per valore come puntatori  */
{int t;
     t = *a;  *a = *b;  *b = t;
}
```

chiamando swap(**&**x, **&**y)
SI OTTIENE L'EFFETTO DESIDERATO

Esempio di passaggio di strutture dati

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

main()
      {  int c[3], d; str str1;
      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);
      str1 = prova(c,d,3);
      printf("Dopo: %d,%d,%d,%d\n",c[0],c[1],c[2],d);
      }

   str prova(int a[ ], int b, int n)
   /*  a per riferimento - b,n per valore  */
   {
      int i; str str1;
      for (i = 1; i < n; i++) a[i] = b;
      b = a[0];
      str1.a = b; str1.b = '\n';
      return str1;
   }
```

Il risultato dell'esecuzione di questo programma è:
Prima: 100,15,20,0
Dopo: 100,0,0,0
e il valore di **d** e **str1?**

PROGRAMMA

Un programma risulta sempre
dalla interazione con un ***sistema operativo***
e andando oltre un *linguaggio di programmazione*

Un programma risulta dall'insieme di

MODULI statici che un linker mette insieme
ciascuno composto di linguaggi anche diversi

BLOCCHI dinamici
che presentano gli algoritmi di soluzione
ad esempio, procedure, funzioni, blocchi

in C

Il **main** è riconosciuto per nome
MODULI statici sono file distinti
solo **FUNZIONI BLOCCHI** dinamici

NON sono possibili innestamenti
di **moduli statici** (file) e
di **blocchi dinamici** (con dichiarazioni funzioni {})
regolando la visibilità

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
- restituisce il carattere letto **convertito in int** o
EOF in caso di end-of-file o errore

int **putchar**(int c); scrive un carattere
- 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
- 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
- in caso di errore restituisce **EOF**

Esempio (lo stesso di prima):

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

- le **stringhe di caratteri** vengono memorizzate **in array di caratteri**
- le stringhe di caratteri **terminano con** il carattere **'\0'** (**NULL** - valore decimale zero)
- la gets **sostituisce** il **new line con** il **NULL**
- 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);
- scrive una serie di valori in base alle specifiche contenute in format
- i valori sono i risultati delle espressioni expr1, expr2, ..., exprN
- restituisce il numero di caratteri scritti, oppure EOF in caso di errore

int **scanf** (char *format, &var1, &var2, ..., &varN);
- legge una serie di valori in base alle specifiche contenute in format
- memorizza i valori nelle variabili var1, var2, ..., varN **passate per riferimento**
- 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.,};
```

ENUMERAZIONE

typedef **enum {**a1, a2, a3, ... , an**}** enumtype;

typedef enum {Gen, Feb, Mar, ...} mesi;
... mesi m1, m2; ...

Definizione delle due costanti booleane

typedef enum {FALSE,TRUE} Boolean;

DEL TUTTO EQUIVALENTE ALLA PRECEDENTE!

Boolean flag1,flag2;
flag1 = TRUE;
flag2 = -37 /* !!! */
if(flag2) ... **funziona lo stesso!**

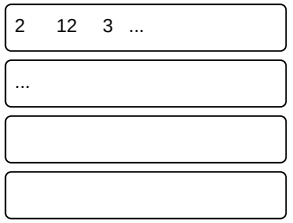
ARRAY

tipoelemento nome [numero elementi]

STRUTTURA DATI COSTANTE
int vettoreinteri [20]; (gli elementi vanno da 0 a 19)
... vettoreinteri **[**15**]** ...

La definizione di array multidimensionali
tipoelemento nome
[numeroelementi1dimensione]
[numeroelementi2dimensione]
...
[numeroelementiNdimensione]

Memorizzazione per righe: int Mat [4] [8], Mat [0]



ESEMPIO

```
typedef int vettore [30];
vettore v;  int i, nontrovato = 1, chiave; ...
for  ( i = 0; i < 30 && v[i] > 0 ; i = i + 1)   v[i]= 10;
i = 0;
while  (i < 30 && v[i] > 0)  {  v[i] = 10;    i= i + 1;  }
 ...
for  ( i = 0; i < 30 && nontrovato ; i = i + 1)
      {if (v[i] == chiave) nontrovato = 0;}
```

ESEMPIO DI MATRICE

In genere, si usano
struttura dati e
strutture di controllo corrispondenti

```
int i, j, k;
int M1 [10][20], M2[20][30], M3[10][30];

for (i=0; i < 10; i=i+1)          /* inizializzazione di M3 */
    for (j=0; j < 30; j=j+1) M3[i][j] = 0;

            /* ripetizioni enumerative */


/* prodotto di matrici M1 e M2 in M3 */
for (i=0; i < 10; i=i+1)
    for (j=0; j < 20; j=j+1)
        for (k=0; k < 30; k=k+1)
M3[i][k] = M3[i][k] + M1[i][j] * M2[j][k];
```


ARRAY

Esempio di **inizializzazione di un array**

```
int v[10] = {1,2,3,4,5,6,7,8,9,10};
/*  v[0] = 1; v[1] = 2; ... v[9] = 10;  */

int v[ ] = {1,2,3,4,5,6,7,8,9,10};
```

Memorizzazione per righe
l'indice più a destra varia più velocemente

matrix

	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3	3,0	3,1	3,2	3,3

Inizializzazione di array multi dimensionale

```
int matrix[4][4] = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}};

int matrix[ ][4] ={{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}};
```

Esempi

int vector [30];

Il passaggio alle funzioni sempre **per riferimento**

```
... fun(vector,...) ... /* INVOCAZIONE  */
... fun(&vector,...) ... /*  del tutto equivalente  */

fun(vector[4],...) { ... }      /*DEFINIZIONE  */
fun(vector[ ],...) { ... }
```

la (prima) dimensione può essere omessa nella definizione di funzione
le altre sono necessarie e **devono essere costanti**

```
fun(int n1,int n2,int m[n1][n2],...) { ... }  /*  NO!  */
```

NON SI POSSONO USARE VARIABILI
PER LE DIMENSIONI DEI VETTORI

NON c'è controllo di range

```
...vector[2]   ...vector[32] ...vector[-1]  ...vector[450]
/* tutte corrette  */
```

Un array è sempre passato come puntatore all'elemento

```
... fun(vector,...) ...          /* INVOCAZIONE  */
... fun(vector[2],...) ...       /* scorretta  */
... fun(&vector[2],...) ...      /* corretta  */
... fun1(matrix[2][2],...)
... fun2(&matrix[2][2],...)
... fun3(matrix[2],...) ...
```


STRUCTURE

struct { lista dichiarazioni elementi }
in analogia con un record

```
struct       {  int anno;
                   int mese;
                   int giorno;
                } date;
 ... date. anno ...

typedef struct  {    int anno;   int mese;   int giorno;
                   } datatype ;
 datatype  data1;

  ... data1. anno ...
```

UNION

union { definizione 1; definizione 2; ... **}**
analogo alla parte variante di record in Pascal
tutti i componenti condividono la stessa memoria

```
union {   float raggio;
          int   latirettangolo [2];
          int   latitriangolo  [3];
       } oggetto;

  oggetto. latirettangolo [0]
```

STRUTTURE A BIT

Definizione di strutture a bit

```
struct        {
   identificatore1: ampiezzaCampo1;
   identificatore2: ampiezzaCampo2;
    ...
   identificatoreN: ampiezzaCampoN
                    }
```

Ogni campo identifica un **insieme dei bit** della struttura
===> **stretta dipendenza** dalla architettura
(*a partire dal bit alto o basso*)

```
typedef struct                {
      CarryFlag      : 1;     Dont1          : 1;
      ParityFlag     : 1;     Dont2          : 1;
      AuxFlag        : 1;                    : 1;
      ZeroFlag       : 1;     SignFlag   : 1;
      TrapFlag       : 1;     IntFlag        : 1;
      DirecFlag      : 1;     OverfFlag  : 1;
                              : 4; } Flagt;
Flagt Flags, *FlagPtr;

if ( Flags. OverfFlag )    <overflow>;
```


PUNTATORI

Definizione di una variabile puntatore

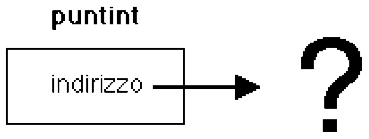
tipoElementoPuntato *nomePuntatore

Esempio:

```
int  k,*puntint, x;     x = 5;
```

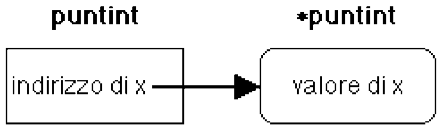
puntint è il puntatore ➠ contiene l'indirizzo dell'elemento puntato

*puntint è l'elemento puntato ➠ contiene il valore (intero) dell'elemento puntato



OPERATORE INDIRIZZO

```
puntint = &x;        /*  &x e' l'indirizzo di x  */
```




```
k = *puntint;            /*  k = 5  */
*puntint = k + 1;        /*  x = 6  */
```

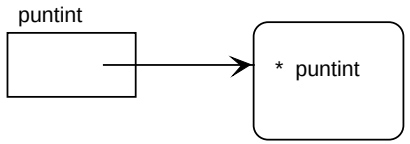
UGUAGLIANZA vs. IDENTITÀ

relazione	relazione
sui puntati	sui puntatori

ALLOCAZIONE DINAMICA

void * **malloc**(size_t size);
alloca un blocco di size byte in memoria centrale (nell'area **heap**)
restituisce l'**indirizzo del primo byte**, oppure **0** in caso di mancanza di spazio

void **free**(void *block);
dealloca un blocco di memoria di indirizzo iniziale block, allocato con una precedente chiamata alla malloc



non sono definite
a livello di **linguaggio di programmazione**
ma a livello di **sistema operativo**

Esempio:

```
int *p;
   ... p non è ancora definito ...
p = (int *) malloc(sizeof (int));
   ... p è definito, il suo contenuto non ancora ...
*p = 55; /* posso scrivere p[0] = 55 ? */
   ... p e *p sono definiti e utilizzabili ...
free(p);
   ... p non è più definito ...
```


Problemi connessi con i puntatori

DANGLING REFERENCES

Possibilità di fare riferimento ad aree di memoria non più allocate al programma

```
int *p;
    p = (int *) malloc(sizeof(int));
    ...
    free(p);
    ... *p ...      /* Da non fare! */
```

AREE INUTILIZZABILI

Possibilità di perdere il riferimento ad aree di memoria allocate al programma (non più riusabili)

```
int *p1,*p2;
    p1 = (int *) malloc(sizeof(int)); *p1 = 10;
    p2 = (int *) malloc(sizeof(int)); *p2 = 20;
    *p1 = *p2;    /* SI: *p1 == *p2 == 20 */
    p1 = p2;      /* Da non fare! */
```

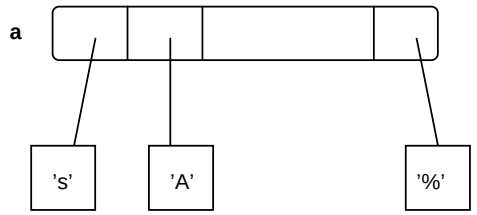
L'area puntata da p1 non è più raggiungibile, ma rimane allocata al programma!

NOTA PRECEDENZA *

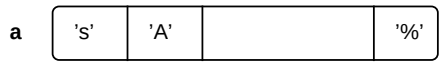
[ ] HA PRECEDENZA RISPETTO A ***

Quindi **char *a[ ]**; ===> equivale a
char *(a[ ]);

a è un *array di puntatori a caratteri*.



Per un *puntatore ad un array di caratteri*
è necessaria la parentesi
char (* a) [ ]



Conversioni di tipo

I linguaggi che fanno delle conversioni automatiche sono poco sicuri

Conversioni implicite (⇐) e
promozioni di tipo (⇑) nelle espressioni:

double ⇐ float
⇑
unsigned long
⇑
long
⇑
unsigned int ⇐ unsigned short
⇑
int ⇐char, unsigned char, short, enum

Conversioni esplicite (CASTING)

(nomeTipo) espressione

Esempio:

```
int  v1,v2;
float x,y;
v1 = 5; v2 = 2;
x = v1 / v2;                    /*  x = 2.0  */
y = (float) v1 / (float) v2;    /*  y = 2.5  */
```

Equivalenza di tipo

STRUTTURALE Variabili con la **stessa struttura interna** sono **equivalenti** e quindi **assegnabili** tra loro

PER NOME Sono equivalenti solo variabili che fanno riferimento alla **stessa definizione di tipo**

*In **Pascal** equivalenza per nome*

In **C** non si specifica quale equivalenza
Implementazioni con equivalenza strutturale
--- **usare sempre equivalenza per nome** ---

```
typedef struct S1 {  int x,y,z;  } T1;
T1 s1,s2;  struct S1 s3;
typedef struct S2 {  int x,y,z;  } T2;
T2 h1,h2;  struct S2 h3;
```

s1,s2,s3,h1,h2,h3 sono tutti equivalenti per struttura
s1,s2 e s3 sono equivalenti per nome
h1,h2 e h3 sono equivalenti per nome
s1 e h1 non sono equivalenti per nome

```
s1 = s3;      /*  SI!  */
h1 = s3;      /*  NO!  */
```


OPERAZIONI SU STRUTTURE DATI

per i dati costruiti possiamo considerare le operazioni fondamentali di assegnamento e confronto (identità)

Tipo di dato strutturato	Assegnamenti	Confronti
array (costante)	no	no
strutture	si	no
unioni	si	no
puntatori a dati	si	si
puntatori a funzioni	si	si
puntatore a dati e puntatore a funzione (e viceversa)	no	no
puntatore a dati e array	si	no
array e puntatore a dati	no	no

Il **puntatore** è un costruttore **molto 'libero'**
pericolosamente vicino all'indirizzo

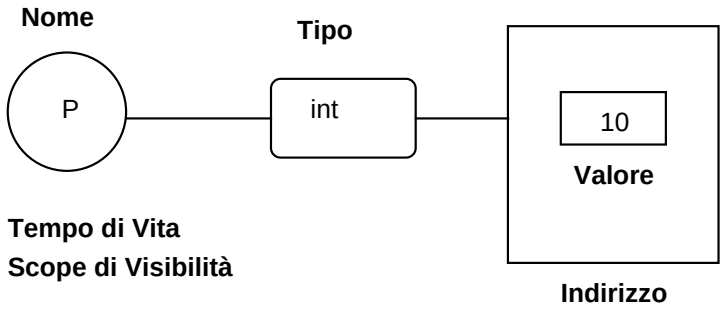
Vale il concetto di uguaglianza (dei puntati)

PROGRAMMA

Struttura di un programma (1 file)

```
** inclusione header file per librerie standard C
#include <stdio.h>
# ...
** definizione variabili globali all'intero programma
tipoVar nomeVar,  ...;
...
** dichiarazione prototipi funzioni
tipo1 F1(parametri);
...
tipoN FN(parametri);
** definizione della funzione main
main(){
    • definizione variabili locali al main
    • codice del main
    }
** definizione della generica funzione Fj
tipoj Fj(parametri)     {
    • definizione variabili locali alla funzione
    • codice della funzione Fj
        }
```


Ogni **entità** descritta in un programma



Ogni **variabile** ha attributi:

- **Nome**, identificatore unico nel programma
- **Tipo**, per indicare l'insieme dei valori
- **Valore**, tra quelli ammessi dal tipo
- **Indirizzo**, riferimento alla memoria che lo contiene
- **Tempo di vita**, durata di esistenza nel programma
  in relazione al blocco racchiudente
- **Scope**, visibilità del nome nel programma
  in relazione al modulo/blocco di definizione

I linguaggi di alto livello nascondono la
realizzazione fisica (indirizzo)

BINDING legame del nome con la risorsa fisica che la contiene e rappresenta

Ricerca del minimo e massimo di un insieme

```
#define N 15
typedef int vettore[N];
                    /* dichiarazione di due funzioni */
int minimo (vettore vet);   int massimo (vettore vet);

main ()
{ int i;  vettore a;
      printf ("Scrivi %d numeri interi\n", N);
      for (i = 0; i < N; i++) { scanf ("%d", &a[i]); }
      puts ("L'insieme dei numeri è: ");
      for (i = 0; i<N; i++) { printf(" %d",a[i]); }puts("\n");
      printf ("Il minimo vale %d e il massimo è %d\n",
            minimo(a), massimo(a));
}

int minimo (vettore vet)
   {     int i, v, min;
      for  (min = vet[0], i = 1; i < N; i ++)
            {v = vet[i]; if (v < min) min = v; }
      return min;
   }

int massimo (vettore vet)
   {     int i, v, max;
      for (max = vet[0], i = 1; i < N; i ++)
            {v = vet[i]; if (v > max) max =v;}
      return max;
   }
```


Ricerca del minimo e massimo di un insieme

```
#define N 15
typedef int vettore[N];  (versione C Kernigham Ritchie)
                    /* dichiarazione di due funzioni */
int minimo (vet); int massimo (vet);

main ()
{ int i;  vettore a;
      printf ("Scrivi %d numeri interi\n", N);
      for (i = 0; i < N; i++) { scanf ("%d", &a[i]); }
      puts ("L'insieme dei numeri è: ");
      for (i = 0; i<N; i++) { printf(" %d",a[i]); }puts("\n");
      printf ("Il minimo vale %d e il massimo è %d\n",
            minimo(a), massimo(a));
}

int minimo (vet)
vettore vet;
   {     int i, v, min;
      for  (min = vet[0], i = 1; i < N; i ++)
            {v = vet[i]; if (v < min) min = v; }
      return min;  }

int massimo (vet)
vettore vet;
   {     int i, v, max;
      for (max = vet[0], i = 1; i < N; i ++)
            {v = vet[i]; if (v > max) max =v;}
      return max; }
```

Ricerca del minimo e massimo di un insieme

```
#define N 15
typedef int vettore[N];
vettore vet;   /* variabile globale visibile globalmente */
int i;
int minimo (void);      int massimo (void);

main (void)
{ int i;  vettore a;
      printf ("Scrivi %d numeri interi\n", N);
      for (i = 0; i < N; i++) { scanf ("%d", &vet[i]); }
      puts ("L'insieme dei numeri è: ");
      for (i = 0; i<N; i++) { printf(" %d",vet[i]); }puts("\n");
      printf ("Il minimo vale %d e il massimo è %d\n",
            minimo(), massimo());
}

int minimo (void)
  {     int i, v1, min;
      for  (min = vet[0], i = 1; i < N; i ++)
            {v1 = vet[i]; if (v1 < min) min = v1; }
      return min;
  }

int massimo (void)
  {     int i, v1, max;
      for (max = vet[0], i = 1; i < N; i ++)
            {v1 = vet[i]; if (v1 > max) max =v1;}
      return max;
  }
```


Esercizio: Ricerca del minimo e del massimo di un insieme di valori reali

```
#include <stdio.h>
#define NMAX 50

leggi(float vet[ ],int nmax);
void scrivi(float vet[ ],int n);
float minimo(float vet[ ],int n);
float massimo(float vet[ ],int n);

main()
{
int n;
      float a[NMAX];

for (;;)
      {
      n = leggi(a,NMAX);
      if(n == 0) break;
      scrivi(a,n);
      printf("\nIl minimo e' %f ed il massimo e' %f\n",
         minimo(a,n),massimo(a,n));
      }
}
```

```
leggi(float vet[ ],int nmax)
/*
        Legge n valori reali (0 <= n <= nmax),
      li memorizza nelle prime n posizioni di vet e
      restituisce il valore di n
*/
{
int n,i;

for (;;)
      {
      printf("\nNumero di valori reali (<= %d)  ",nmax);
      n = 0; scanf("%d",&n);
      if(0 <= n && n <= nmax) break;
      printf("\nValore errato %d\n",n);
      }
for (i = 0; i < n; i++)
      {
      printf("\nValore numero %d?  ",i);
      scanf("%f",&vet[i]);
      }
return n;
}
```



```
void scrivi(float vet[ ],int n)
{
int i;

printf("\nL'insieme dei valori e':\n");
for (i = 0; i < n; i++)  printf("%f   ",vet[i]);
}

float minimo(float vet[ ],int n)
{
int i;
float min = vet[0];

for (i = 1; i < n; i++)
      if(vet[i] < min) min = vet[i];
return min;
}

float massimo(float vet[ ],int n)
{
int i;
float max = vet[0];

for (i = 1; i < n; i++)
      if(vet[i] > max) max = vet[i];
return max;
}
```

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

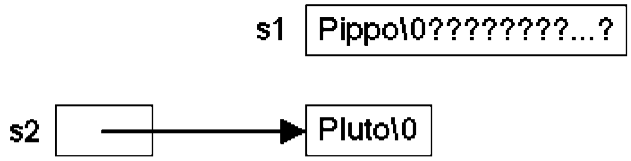
puntatori

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

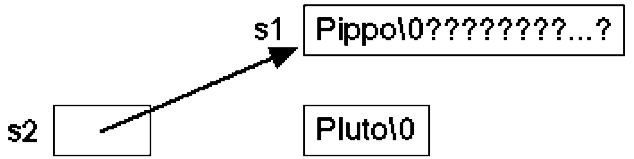
Esempio:

```
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);
```

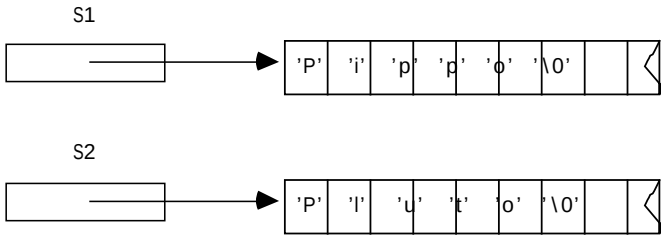
Potremmo **assegnare** da una all'altra, senza problemi

In realtà, il nome di un array è una costante

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

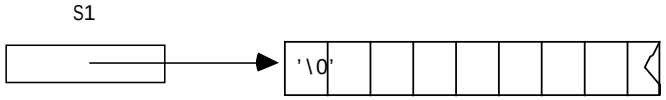
Stringhe

char s1[81];
char *s2;

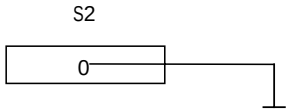


Distinguiamo tra memoria allocata
e lunghezza della stringa contenuta

Casi degni di nota



Stringa VUOTA

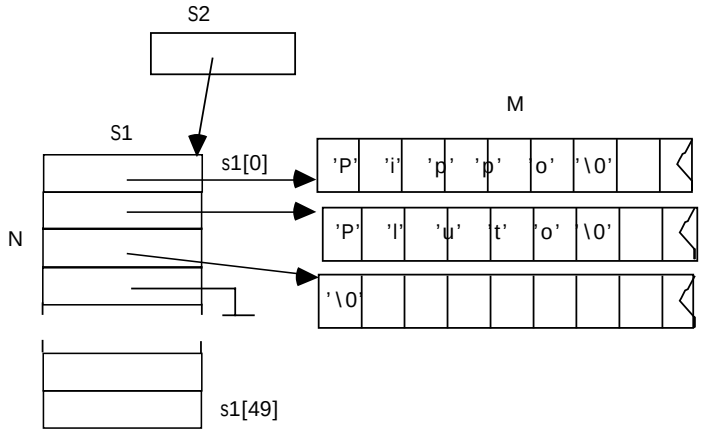


Stringa NULLA

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);
}
```

Argomenti della funzione main

La funzione di startup chiama la funzione **main** con **argomenti argc**, **argv** (ed env)

```
main()
main(int argc,char *argv[ ])
```

- argcè il numero di parametri inseriti sulla linea di comando a livello di S.O.
- argvè un array di puntatori a stringhe di caratteri
  - argv[0] è il nome completo del programma
  - argv[1] è la prima stringa nella linea di comando dopo il nome del programma
  - argv[2] è la seconda stringa nella linea di comando
  - argv[argc-1] è l'ultima stringa nella linea di comando
  - argv[argc] contiene NULL (stringa nulla)

Ad esempio:

```
C:> prova  arg1  "arg con bianchi"  3  4  ultimo!
```

Dove è stato allocata la memoria per le stringhe?
Parte di sistema operativo

Il file "PROVA.C" contiene il seguente programma:

```
#include <stdio.h>
main(int argc,char *argv[ ])
  {
  int k;
  printf("Valore di argc: %d\n",argc);
  for (k = 0; k < argc; k++)
     printf("\targv[%d]: %s\n", k, argv[k]);
  }
```

richiamando il programma con:

```
C:> prova  arg1  "arg con bianchi"  3  4  ultimo!
```

si ottiene:

```
Valore di argc: 6
    argv[0]: prova
    argv[1]: arg1
    argv[2]: arg con bianchi
    argv[3]: 3
    argv[4]: 4
    argv[5]: ultimo!
```

Strutture dati ricorsive

STRUTTURA RICORSIVA

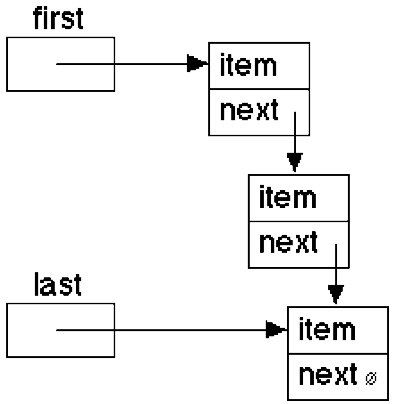
```
#define nil 0

typedef struct node /* structure TAG */
        { char elemento;
          struct node *left, *right;
        } NODE;

typedef NODE * nodePtr;
nodePtr root;

/* allocazione */
root = (nodePtr) malloc (sizeof (NODE));

root -> elemento = 'a';
root -> left = nil;  root -> right = nil;
```

Accesso ai sottocomponenti in modo
diretto e **sequenziale**

```
root -> right =  (nodePtr) malloc(sizeof (NODE));
root -> right -> elemento = 'z';
root -> right -> left = nil;
root -> right -> right = nil;
...
```


Esempio 1

```
#define NULL 0

typedef struct node {
        int item;
        struct node *next;
        } Nodo;
Nodo  *first,*last,n1,n2,n3;
first = &n1;
n1.item = 100; n1.next = &n2;
n2.item = 200; n2.next = &n3;
n3.item = 300; n3.next = NULL;
last = &n3;
```



Di norma ALLOCAZIONE DINAMICA delle strutture!

Esempio 2

```
typedef struct node {
      char item;
      struct node *left,*right;
      } Nodo;

main()
      {
      Nodo  *root = NULL;
      /*  Allocazione del primo nodo  */
      root = (Nodo *) malloc(sizeof(Nodo));
      root->item = 'a';
      root->left = root->right= NULL;
      ...
      root->right= (Nodo *) malloc(sizeof(Nodo));
      root->right->item = 'b';
      root->right->left = root->right->right= NULL;
      ...  creazione dell'albero  ...
      visita(root);
      ...
      }

void visita(Nodo *r)
      {
      if(r == NULL) return;
      printf("%c",r->item);
      visita(r->left);
      visita(r->right);
      }
```


INNESTAMENTO DI STRUTTURE DATI

ESEMPIO DI STRUTTURA
record ed accesso sequenziale

```
#include <string.h>
typedef struct { char nome[10];
                 char indirizzo[20];
                 int  eta;
               } persona;

persona p1, p2;

/* accesso in sequenza  ai campi */
strcpy (p1.nome, "Enrico");
strcpy (p1.indirizzo, "Viale Risorgimento");
p1.eta = 20;

p2. ...     ;

if (  !strcmp(p1.nome, "Enrico")                    &&
      !strcmp(p1.indirizzo,"Viale Risorgimento")  &&
      (p1.eta > valore)
   )   < statement >;
```

Naturalmente si prevede anche l'accesso diretto ai sottocampi della sequenza

Si noti come si **devono** trattare le **stringhe**

ESEMPIO DI UNIONE

```
typedef enum {rettangolo, cerchio, triangolo} figure;

typedef struct   {     int perimetro;
                              figure kind;
        /* sottocampo tag per distinguere i diversi casi*/
                              union { int latirett [2];
                                            int raggio;
                                            int latitr [3];
                                          } kindfig;
                    } figura;
figura f;
f.kind = cerchio;   /* f e' un cerchio */

switch  (f.kind)
{    case cerchio:    f.kindfig.raggio = 10;    break;
     case rettangolo:     < due lati > ;          break;
     case triangolo:      <  tre lati >;               break;
}
```

Manteniamo **esplicitamente** la **consistenza** della struttura dati complessiva

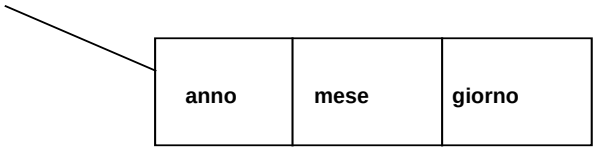
INNESTAMENTO delle **STRUTTURE DATI**

Le strutture si possono innestare

Puntatore ad una **struttura**

accesso ai sottocampi tramite l'operatore **->**

```
typedef   struct   { int anno; int mese; int giorno;
                        }  datatype;

typedef    datatype   *puntdata;
                  /* tipo puntatore alla struttura precedente */
       ...
       puntdata  puntatore;


/* fase di allocazione dell'area puntata */
         puntatore =  malloc ( sizeof (* puntdata));
```




```
...  puntatore -> anno   ...
...  puntatore -> giorno ...
```

INNESTAMENTO STRUTTURE DATI

ESEMPIO DI
PUNTATORE A UNA STRUTTURA

```
#include <string.h>
typedef struct
      {   char nome[10];
          char indirizzo[20];
          int  eta;
      } persona;

typedef persona * perPtr;
       /* puntatore a record */
perPtr    ptr;

/* allocazione */
ptr = (perPtr) malloc (sizeof (persona) );

strcpy (ptr -> nome, "Enrico");
strcpy (ptr -> indirizzo, "Viale Risorgimento");
ptr -> eta = 20;
```

Anche l'accesso può avvenire in modo
diretto e **sequenziale**

INIZIALIZZAZIONE DELLE STRUTTURE DATI

Una struttura dati può essere inizializzata alla definizione (e non alla dichiarazione naturalmente)

Ancora l'esempio di un elemento di un albero

```
typedef  struct  nodo_albero     {
            int                          item;
            struct nodo_albero *left;
            struct nodo_albero *right;
                                         }
                    nodo_albero ;

nodo_albero  nodo = {0, NULL, NULL};
```

dove NULL è definito da
`#define NULL 0`

```
 struct   nodo_albero          {
            int                                item;
            struct nodo_albero      *left;
            struct nodo_albero      *right;
                                    };

struct  nodo_albero  nodo = { 0, NULL, NULL};
```

Definizione di strutture complesse

*a[ ] (*a)[ ] *(a[ ]) *a( ) (*a)( ) *(a( ))

[ ] e () hanno precedenza rispetto a * - quindi:

int *a[ ] e **int *(a[ ])** coincidono è definiscono
un **array di puntatori a int**
int (*a)[ ] è un **puntatore a un array di int**
int *a() e **int *(a())** coincidono è definiscono
una **funzione che restituisce un puntatore a int**
int (*a)() è un **puntatore a una funzione che restituisce un int**

Esempio:

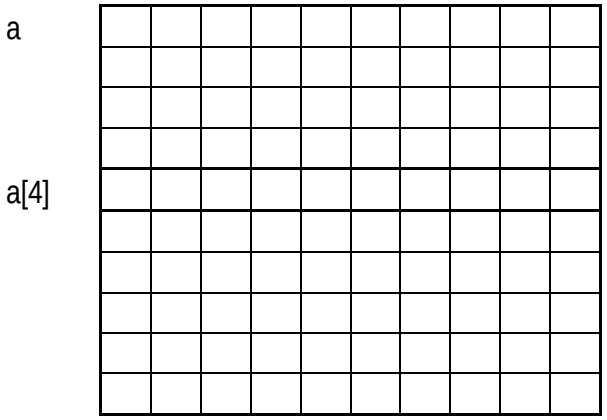
```
char *nome[ ] = {"",
   "Gennaio","Febbraio","Marzo","Aprile",
   "Maggio","Giugno","Luglio","Agosto",
   "Settembre","Ottobre","Novembre", "Dicembre"};

char *nomeMese(int mese)
   {  return (1 <= mese && mese <= 12) ?
      nome[mese] : nome[0];  }

...  printf("%s",nomeMese(3)) ...
```


Esercizio:

```
int a[10][10],*b[10],(*c)[10];
```

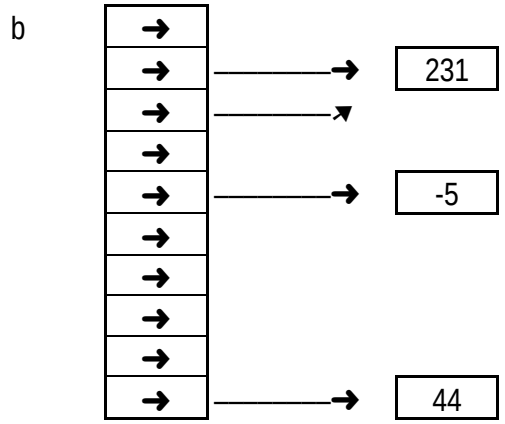
qual è la differenza tra **a**, **b** e **c**?

a è una **matrice 10 x 10** di int
sizeof(a) è 200 == 10 x 10 x sizeof(int)
sizeof(a[k]) è 20 == 10 x sizeof(int)
sizeof(a[k][m]) è 2 == sizeof(int)




```
...  a =  ...          /*  NO!  */
...  a[k] =  ...  /*  NO!  */
```

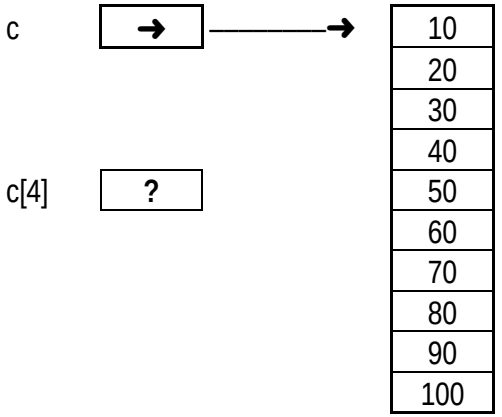
b è un **array** di **10 puntatori** a int
sizeof(b) è 20 (o 40) == 10 x sizeof(int *)
sizeof(b[k]) è 2 (o 4) == sizeof(int *)
sizeof(*b[k]) è 2 == sizeof(int)



... b = ... /* NO! */
... b[k] = ... /* SI! */

```
b[1] = (int *) malloc(sizeof(int));  *b[1] = 231;
b[2] = b[1];    /*  Attenzione!  */
b[4] = (int *) malloc(sizeof(int));  *b[4] = -5;
b[9] = (int *) malloc(sizeof(int));  *b[9] = 44;
```


c è un **puntatore** a un **array di 10** int
sizeof(c) è 2 (o 4) == sizeof(int *)
sizeof(*c) è 20 == 10 x sizeof(int)
sizeof((*c)[k]) è 2 == sizeof(int)



... c = ... /* SI! */
... c[k] ... /* NO! */
... *c[k] ... /* NO! */
... (*c)[k] ... /* SI! */

```
c = (int *) malloc(10 * sizeof(int));   o meglio:
c = (int (*)[10]) malloc(sizeof(int [10]));
(*c)[0] = 10;
   for (k = 1; k <= 9; k++)  (*c)[k] = (*c)[k-1] + 10;
```

ATTENZIONE ALLE PRECEDENZE

Chiamata a procedura ()   Selezioni [ ] -> .
Unari ! ~ + – ++ –– & * (type) sizeof

```
    int *p;
```
qual è la differenza tra
```
    *p++    *(p++)    (*p)++
```

Gli operatori ++ e * hanno la **stessa precedenza** e
sono **associativi da destra a sinistra**

***p++** coincide con ***(p++)** e vuol dire:
usa il valore puntato da p, quindi incrementa p
(*p)++ vuol dire:
usa il valore puntato da p, quindi incrementa tale valore - p rimane immutato

```
void incrementa(int n,int *p)
  {  int k;
     for (k = 0; k < n; k++)  p[k]++;
  }

void incrementa(int n,int *p)    /*  Versione II  */
  {while (n– –  >  0)  (*p++)++;}
```


FUNZIONI: PASSAGGIO dei PARAMETRI

Passaggio per copia per
qualunque tipo (anche costruito)
eccetto array e funzioni

PASSAGGIO del RISULTATO

Parametri di uscita:
qualunque tipo (anche costruito)
eccetto array e funzioni
MA sono possibili **struct** (di qualunque dimensione)

Uso della **semantica per riferimento**
al posto di qualunque tipo (anche costruito)
si usa il **puntatore corrispondente**

FUNZIONI COME PARAMETRI

Il C riconosce le funzioni come **entità del linguaggio**:
le funzioni sono trattate in modo completo ==>
si possono definire il *tipo funzione* e *variabili relative (?)*
si possono passare funzioni in funzioni
- come *parametri di ingresso*
- come *parametro di uscita*

TIPO FUNZIONE

Ogni funzione ha una propria **dichiarazione/definizione**
==>
dichiarazione implicita del tipo corrispondente
double *sin* (double); ==> typedef double funzione (double);
double *fun* (double x) { ... }
Le due funzioni hanno lo stesso tipo (strutturale)

IN C

NON si possono definire variabili di tipo funzione,
solo **definizione/dichiarazioni implicite**

Si possono invece definire variabili del tipo
puntatore a funzione
Il C **assimila** la **funzione** al **puntatore a funzione**

typedef double (* ptrfunzione) (double);
ptrfunzione v1ptr;

Variabili di questo tipo possono essere usate
in **assegnamenti**

v1ptr = & sin;
/* anche ***v1ptr = sin;*** */

PASSAGGIO di parametro e ASSEGNAMENTO ➠
NON VALUTAZIONE
si riferisce la funzione attraverso il nome della funzione

VALUTAZIONE
La invocazione della funzione è sempre seguita dalle parentesi (e dai parametri effettivi)

PASSAGGIO DI FUNZIONI COME PARAMETRI DI INGRESSO in FUNZIONI

DEFINIZIONE/DICHIARAZIONE funzione

```
double fun (double x);

double sommaquadratif
      (double (* f) (double par), int m, int n)
      /*  anche (double (f) (double par), int m, int n)  */
```

INVOCAZIONE

```
sommaquadratif (&fun, 1, 10000)
/*  anche sommaquadratif (fun, 1, 10000)  */
```

USO del parametro

```
somma = somma + (* f)(k) * (* f)(k);
/*  anche somma = somma + (f)(k) * (f)(k);  */
```

- Priorità **()** rispetto a *****

(double (f) (double par), int m, int n)*
f punta alla funzione con un parametro di ingresso double e restituisce un double

- Il **nome della funzione** è assimilato
al **puntatore alla funzione** stessa
(come astrazione dell'indirizzo del suo codice)

Esempio

```
   double fun ( double x)     /* funzione reciproco */
       {  return 1.0 / x;}
   double sin (double);       /* funzione seno di libreria */

double sommaquadratif (double f ( double par),
                       int m, int n)
 { int k;  double somma;
 somma = 0;
 for (k=m; k <= n; k++)
          somma = somma + f(k) * f(k);
 return somma;
   }

double sommacubif (double (* f) ( double par),
                       int m, int n)
 { int k;  double cubo;
 cubo= 0;
 for (k=m; k <= n; k++)
          cubo = cubo + (* f)(k) * (* f)(k) * (* f)(k);
 return cubo;
 }

main ()
{  int a, b, c, i;
  printf (" Inversi %.7f\n",sommaquadratif (fun,1, 10000));
  printf (" Seni %.7f\n",  sommaquadratif (sin, 2, 13));
  printf (" Inversi %.7f\n", sommacubif (fun, 1, 10000));
  printf (" Seni %.7f\n", sommacubif (sin, 2, 13));
}
```


Esempio di passaggio di funzione

void **qsort** (void *base,size_t nelem,size_t width,
int (***fcmp**)(void *elem1,void *elem2));

- **qsort** è una **funzione generica di ordinamento** (algoritmo quicksort) - libreria C - prototipo nel header file stdlib.h
- **base** punta allo 0-esimo elemento dell'array da ordinare
- **size_t** è un tipo definito in stdlib.h (in genere un unsigned long)
- **nelem** è il numero di elementi nell'array
- **width** è la lunghezza di ogni elemento dell'array in byte
- **fcmp** è la funzione di paragone che deve essere definita dal programmatore e che deve essere passata alla qsort
  - i due **argomenti** sono i puntatori ai due elementi dell'array da confrontare
  - il **risultato** deve essere il seguente:
    *elem1 < *elem2 un intero < 0
    *elem1 == *elem2 0
    *elem1 > *elem2 un intero > 0

Ordiniamo gli n elementi di un vettore di reali arr

- *base* arr (che coincide con &arr[0])
- *nelem* n
- *width* sizeof(arr[0]) oppure sizeof(float)
- *fcmp* deve essere definita

```
#include <stdio.h>
#include <stdlib.h>
#define NMAX 50

int confronta(float *v1,float *v2);
main()
   {
   int n; float arr[NMAX];
   n = leggi(a,NMAX);
   if(n == 0) exit(0);
   qsort(arr,n,sizeof(float),confronta);
   /*  per un migliore controllo del tipo: cast
   (  int (*) (void *elem1,void *elem2)) confronta;
   */
   scrivi(arr,n);
   }

int confronta(float *v1,float *v2)
   {
   if(*v1 < *v2) return -1;
   else if(*v1 == *v2) return 0;
   else return 1;
   }
```


VARIABILI DI TIPO FUNZIONE (!?)

Si possono definire solo variabili di tipo

puntatore a funzione

Le variabili sono usate per riferire funzioni da invocare successivamente

```
typedef int fprot1 (int a, int b);
```
tipo funzione **solo per fasi dichiarative**

```
typedef fprot1 * funptr;
```
tipo puntatore a funzione (anche variabili)

```
typedef fprot1 *   tavola1 [20];
tavola1 tabella1;
typedef funptr     tavola2 [20];
tavola2 tabella2;
```

Gli **elementi delle due tabelle** sono **puntatori a funzioni** con **due parametri** interi di ingresso e che restituiscono un **intero**

```
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; }

for (i=0; i<20; i++)  { if (i && ...) tabella1 [i] = selecta;}   ...
  printf(" Valore %d\n", tabella 1[12] (12, 2)+ 4);
```

FUNZIONI come PARAMETRI di RITORNO

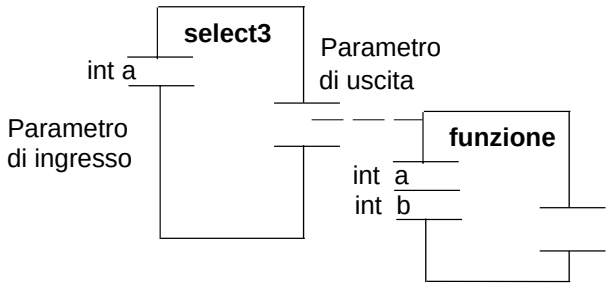
NON si possono fornire direttamente funzioni
come parametri di ritorno

Una funzione **può restituire** un
puntatore a funzione

```
typedef int fprot1 (int a, int b);
typedef fprot1 * funptr;

fprot1 * select1 (int a) ...
```
La funzione select1 ritorna un puntatore a funzione del tipo a due parametri interi e un intero di ritorno

anche select2
```
funptr  select2 (int a) ...
```

e anche select3
```
int (* select3 (int a)) (int a, int b) ...
```



STRUTTURE DATI PIÙ INNESTATE

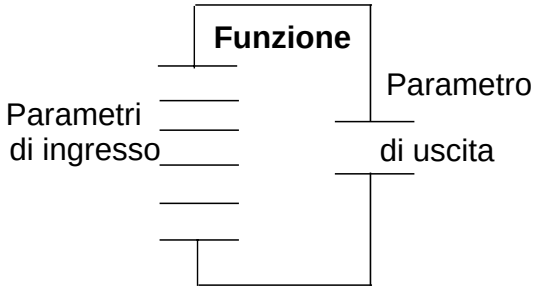
```
typedef int         fprot1 (int a, int b);
typedef fprot1 *    funptr;
typedef funptr (*   tipofunz) (int a);

typedef funptr (*  tavfunzionali [10]) (int a);
```
definizione di un tipo array di puntatori a funzione

Le funzioni ritornano il puntatore a funzione di tipo fprot1

```
funptr  select2 (int a);
int selecta (int a, int b);
int selectb (int a, int b);
int selectc (int a, int b);

main ()
{     int a, b, c, i;
      tipofunz varfun;
      tavfunzionali tabella;

varfun = select2;
for (i=0; i<10; i++)
   printf(" valore funzione %d\n", varfun (i) (1, 2) );
/* sono in gioco due valutazioni di funzione */

for (i=0; i<10; i++) tabella [i] = select2;
for (i=0; i<10; i++)
   printf(" valore ricavato dalla tabella %d\n",
      (* tabella [i])(i) (i,i));
}
```

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**

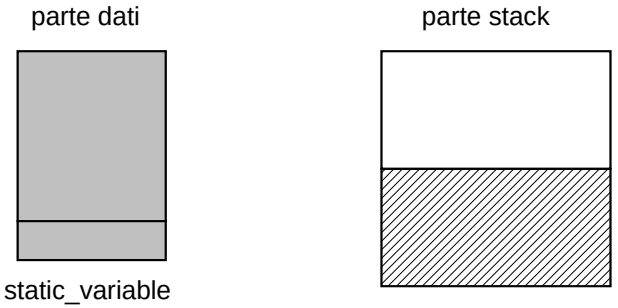
```
int *ptr;
ptr = (int *) malloc (sizeof (int));
* ptr = 55;
```


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

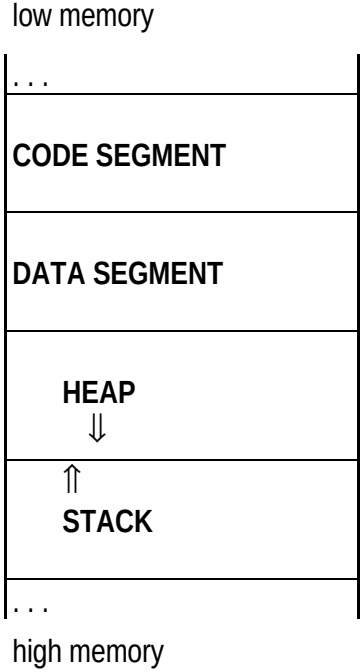
TEMPI di VITA
VISIBILITÀ

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

- come le auto
- 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

APPLICAZIONE SU PIÙ FILE

Compilazioni indipendenti dei file + **collegamento**

Durante la compilazione di un file sorgente, il compilatore **non vede le entità** (variabili e funzioni) **definite negli altri file** ➠ è necessario **dichiarare le entità esterne** utilizzate

DICHIARAZIONE: fornisce l'**interfaccia** a una
funzione (prototipo), **dato** o **tipo di dato**
--- non viene allocato spazio in memoria

```
extern fattoriale(int n);
extern float xyz(...);    /*  in altro modulo  */
extern int ncall;    /*  in altro modulo  */
typedef short int Signed16;
```

DEFINIZIONE: fornisce l'**implementazione** di una
funzione o **dato**
--- viene allocato spazio in memoria

```
fattoriale(int n) {...}
int ncall = 0;        /*  globale  */
```

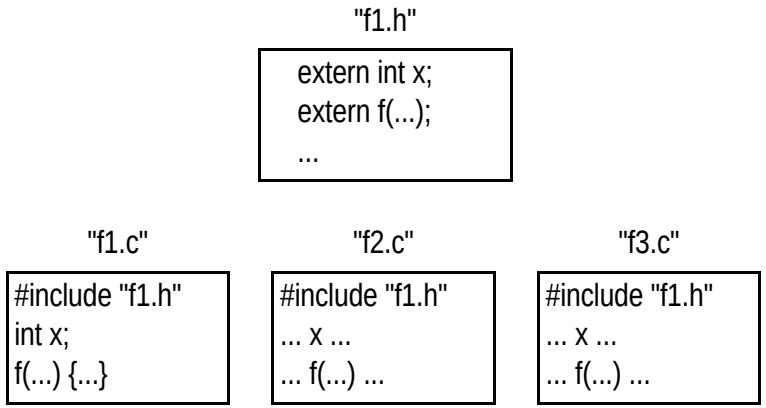
Una DEFINIZIONE può fungere anche da DICHIARAZIONE

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

Il file "f1.c" **mette a disposizione** la variabile x e la funzione f() - DEFINIZIONI

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

Tutte le **dichiarazioni** possono essere inserite **in un HEADER FILE** "f1.h" incluso dai file utilizzatori



Un **header file** contiene SOLO **dichiarazioni** e
NON **definizioni**

La inclusione di una libreria

```
#include <stdio.h>
STDIO per SUN

/*@(#)stdio.h 1.2 86/10/07 SMI; from UCB 1.4 06/30/83 */
# ifndef FILE
#define BUFSIZ 1024
#define _SBFSIZ     8

extern struct _iobuf {
      int   _cnt;
      unsigned char *_ptr;
      unsigned char *_base;
      int   _bufsiz;
      short     _flag;
      char      _file;          /* should be short */
}       _iob[ ];

#define _IOFBF 0
#define _IOREAD    01
#define _IOWRT     02
#define _IONBF 04
      ...
#define NULL    0
#define FILE    struct _iobuf
#define EOF     (-1)

#define stdin         (&_iob[0])
#define stdout  (&_iob[1])
#define stderr        (&_iob[2])
```



```
#define  getc(p)        (--(p)->_cnt>=0?
                        ((int)*(p)->_ptr++):_filbuf(p))

#define  getchar()      getc(stdin)

#define  putc(x,p) (--(p)->_cnt>=0?
                  (int)(*(p)->_ptr++=(unsigned char)(x)):
                  _flsbuf((unsigned char)(x),p))
#define  putchar (x)  putc((x),stdout)

#define  feof(p)        (((p)->_flag&_IOEOF)!=0)
#define  ferror(p) (((p)->_flag&_IOERR)!=0)
#define  fileno  (p)    ((p)->_file)

#define  clearerr(p)    (void)
        ((p)->_flag&= ~(_IOERR|_IOEOF))

extern FILE  *fopen();
extern FILE  *fdopen();
extern FILE  *freopen();
extern FILE  *popen();
extern FILE  *tmpfile();
extern long    ftell();
extern char   *fgets();
extern char   *gets();
extern char   *ctermid();
extern char   *cuserid();
extern char   *tempnam();
extern char   *tmpnam();
#define L_ctermid      9     ...
```

Il preprocessore

Il primo passo di sviluppo in C è fatto attraverso una sostituzione di testo sorgente: il ***preprocessore***

Una macro è una istruzione da espandere in un altro testo con *procedimento di sostituzione testuale*

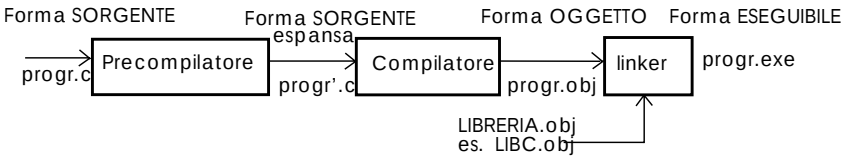
```
#define TRUE   1
#define FALSE  0

#define begin    {
#define end      }
```

```
main ()                          main ()
    begin
     ...                             {...}
    end          ==>
```

Sono possibili anche **macro parametriche**

Il preprocessore lavora prima del compilatore

inserisce i simboli in una tabella delle macro e espande (per ogni token) in termini dei simboli già definiti



DIRETTIVA #define

Definisce una **macro**
Le macro forniscono un meccanismo di **sostituzione testuale con** o **senza parametri**

espansione della macro

Macro senza parametri

#define identificatore [testo]

Ad esempio:

```
#define TRUE  1
#define FALSE  0
#define and  &&
#define or  ||
#define Messaggio "Questo e' un messaggio..."
#define COMPILA
...
if(x and y) puts(Messaggio);
   => if(x && y) puts("Questo e' un messaggio...");
else flag = TRUE;
   => else flag = 1;
...
```

MACRO CON PARAMETRI

#define identificatore(parametri) testo

... **identificatore(parametri attuali)** ...

Durante l'espansione della macro i parametri attuali sostituiscono i parametri formali

Ad esempio:

```
#define Min(x,y)  (x < y ? x : y)
... Min(a,b) ...      /*  produce (a < b ? a : b)  */
```

Che differenza c'è nel considerare la macro con operandi tra **parentesi**?

```
#define Min(x,y)    ((x)>(y) ? (y): (x))
```

Mettere sempre gli **operandi tra parentesi**!

```
#define CUBO(x)   x*x*x
... 3/CUBO(a+b)  ...
   3/a+b*a+b*a+b

#define CUBO(x)   ((x)*(x)*(x))
... 3/CUBO(a+b)  ...
   3/((a+b)*(a+b)*(a+b))
```


ALCUNI ESEMPI DI MACRO:

```
#define ODD(i)                  ((i) & 1)
#define EVEN(n)                 (!ODD(n))  o  (~(n) & 1)

#define DIV2(i)                 ((i) >> 1)
#define MUL2(i)                 ((i) << 1)

#define DEMORGAN1(a, b)    (~((~a) | (~b)))
#define DEMORGAN2(a, b)         (!((!a) || (!b)))
#define COMPLA2(a)              (1 + (~a))
#define AND(a, b)               ((a) ? (b) : 0)
#define OR(a, b)                ((a) ? (a) : (b))

#define NEG(a)                  ((a) ? 0 : 1)
#define ever                    (;;)

#define F1          0x01
#define F2          0x02
...
#define SetFlag(k,n)  ((k) |= (n))
#define ResetFlag(k,n)     ((k) &= ~(n))
#define IsSetFlag(k,n)     ((k) & (n))

int m;
SetFlag(m,F2);          /*  ((m)) |= (0x02))  */
if(IsSetFlag(m,F1)) ...
#define Bit(n)                  (1 << ((n - 1)))
#define SetFlag(k,n)  ((k) |= Bit(n))
#define ResetFlag(k,n)     ((k) &= ~Bit(n))
#define IsSetFlag(k,n)     ((k) & Bit(n))
```

Espansioni ripetute vs. espansioni ricorsive

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

#define mam(x)  (x)+mam(x)

#define sqrt(x) \
( ((x) < 0) ? sqrt(-(x)) : sqrt (x))

#define sum(x,y)      add(y,x)
#define add(x,y)      ((x)+(y))


main()
{ int a, b, c;
 double x = 4.0;
a = 10;
b = 5;
c = 3;


printf("Valore di sum(sum(a,b),c) => \
    %d\n", sum(sum(a,b), c));

printf("Valore di sqrt(-x) ==> \
    %lf\n", sqrt(-x));


/* printf("Valore di mam(a) ===> %d\n", mam(a));
NON SI RIESCE A FARE
IL PREPROCESSORE ESPANDE SOLO (a)+mam(a)
IL COMPILATORE DOPO NON TROVA LA FUNZIONE mam() */
}
```

Vedere cosa produce il preprocessore C (**cpp**)

Sostituzioni TESTUALI

```
#define Swap(xtype,x1,x2)     \
   {xtype xt; xt = x1; x1 = x2; x2 = xt;}
...
int a,b; float x,y; ...
if(a < b) Swap(int,a,b)
else if(x < y) Swap(float,x,y)
```

Possibilità di definire funzioni generiche

ossia che *trattano anche più di un tipo di parametri*

```
#define GenericSwap(funName,elemType)   \
   void funName(elemType *a,elemType *b)   \
      {elemType t; t = *a; *a = *b; *b = t;}
...
GenericSwap(intSwap,int)
GenericSwap(floatSwap,float)
...
int a,b; float x,y; ...
if(a < b) intSwap(&a,&b);
else if(x < y) floatSwap(&x,&y);
```

DIRETTIVA #undef

Cancella la definizione di una macro

#undef identificatore

DIRETTIVE CONDIZIONALI

Le direttive

#ifdef macro-identificatore
#ifndef macro-identificatore
#if espressione-costante
#elif espressione-costante
#else
#endif

permettono di avere programmi diversi (condizionati)

```
#ifndef TRUE
  #define TRUE 1
  #define FALSE 0
#endif
```

ESEMPI:

1) /* FILE PROVA1.C */

```
#include "preproc.h"
#ifndef N
#define N 1
#endif

main ()   {
#if N < 10
int a = N;
#else
int a = 0;
#endif
printf("Valore di a = %d\n", a);
          }
```

/* FILE PREPROC.H */

```
#define N 100
```


2)

/* FILE PROVA2.C */

```
#define versione1

main ()
{#ifdef versione1
int x = 3, y = 5, z;
#else
float x = 3.2, y = 5.4, z;
#endif

z = x * y;

#ifdef versione1
printf("Valore di z = %d\n", z);
#else
printf("Valore di z = %f\n", z);
#endif
}
```

MACRO
SVANTAGGI:

Ci sono alcuni **svantaggi** nell'uso di un preprocessore:

1) un passo in più sul testo sorgente;

2) alcuni errori rilevati dal compilatore e riportati nel sorgente possono non essere chiari per l'intervento del preprocessore

VANTAGGI:

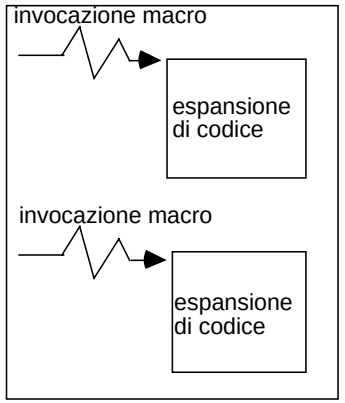
D'altra parte il preprocessore consente di specificare delle **funzioni generiche** nei **tipi di parametri**.

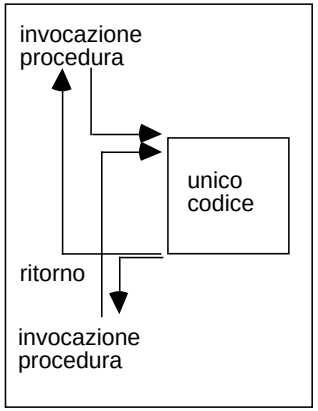
Esempio di funzione generica

```
#define GENERIC_SWAP (NAME, ELEM_TYPE) \
      void NAME (a,  b)\
            ELEM_TYPE *a, *b;\
            { ELEM_TYPE t;\
               t = *a; *a = *b; *b = t; }

Utilizzo:          GENERIC_SWAP (swap_float, float)
Espansione di:     GENERIC_SWAP (swap_int, int)
      void swap_int  (a,  b)
            int *a, *b;  { int  t;
                              t = *a; *a = *b; *b = t;  }
```


MACRO ≠ PROCEDURE

MACRO	PROCEDURE
espansione testuale per ogni invocazione	codice unico
Più ripetizioni	1 solo codice
parametri	parametri
innestamento statico	innestamento dinamico





ESERCIZIO

```
#define TRUE 1
#define FALSE 0

/*  Definizione di due macro
      la prima macro generica espande una procedura di sort:
     sono parametri il nome, il tipo di elementi, il numero,
     e le due funzioni di confronto e scambio  */


#define GENERIC_SORT(fname, type, fcomp, fswap)\
          void fname (v, dimension)\
          type v[]; int dimension;\
          { int i, sc; \
             do {sc = FALSE;\
                  for (i = 0; i < dimension -1; i++)\
                    if (fcomp (v[i], v[i+1])) \
                       { fswap (& v[i], & v[i+1]); \
                          sc = TRUE; }\
                  } while (sc);\
          }


/* macro per funzione generica di scambio di elementi */
#define GENERIC_SWAP (name, elemtype) \
          void name (a, b)\
              elemtype *a, *b; \
          { elemtype t; t = *a; *a = *b; *b =t; }
```



```
/* espansione della macro per scambio elementi */
GENERIC_SWAP    (intswap, int)

int intcomp  ( int a,b)
   { return a - b }

/* espansione della macro per sort di interi*/
GENERIC_SORT   (int_sort, int, intcomp, intswap)

int vett [5] = {10, 5, 4, 3, 11};

main ()
{ int j, i = 5;
   int_sort (vett, i);
}
```

ESPANSIONE del ***MACROPROCESSORE***

```
void intswap (a, b)
      int *a, *b;
{  int t; t = *a; *a = *b; *b =t; }

int intcomp ( int a,b)
{ return a - b }

void int_sort (v,  dimension)
int v[]; int  dimension;
{ int i, sc;
      do {sc = 0;
           for (i = 0; i <  dimension -1; i++)
                    if ( intcomp (v[i], v[i+1]))
                          {  intswap (& v[i], & v[i+1]); sc = 1; }
         } while (sc);
}

int vett [5] = {10, 5, 4, 3, 11};

main()
{
int j, i = 5;
   int_sort (vett, 4);
}
```

Provate a verificare la leggibilità della vera espansione leggendo il codice espanso dal preprocessore

DEFINIZIONE E USO DI MACRO

```
#define  bytealto(w)          ((w) >> 8)
#define  bytebasso(w)     ((w) & 0x00ff)

#define even(n)               ! ((n) & 1)

#define setbit(w,n)          ((w) |=  (1 << n))
#define resetbit(w,n)        ((w) &= ~(1 << n))
#define issetbit(w,n)        ((w)  &  (1 << n))
```

Scrivere le macro:

lowzeroes(n) parola con n bit meno significativi a 1
lowones(n) parola con n bit meno significativi a 1

middlezeroes(n, how) parola con bit tutti ad 1 eccetto le posizioni a partire dal bit n, per how bit a 0

middleones(n, how) parola con bit tutti ad 0 eccetto le posizioni a partire dal bit n, per how bit ad 1

INCLUSIONE DI FILE SORGENTI

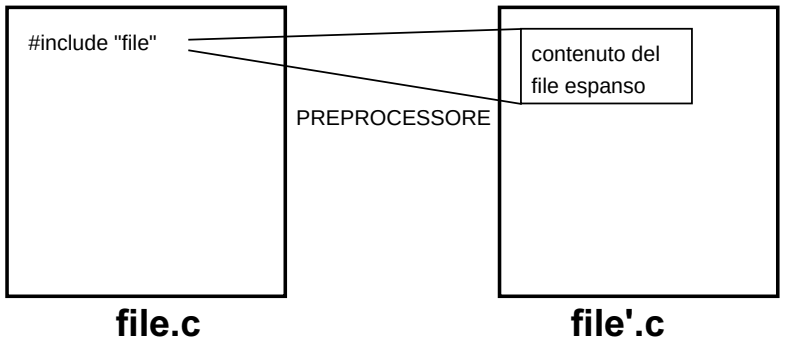
un file sorgente può essere incluso nella sua forma sorgente da tutti i file che devono **condividere** le stesse definizioni:

#include NomedelFiledaIncludere

La ricerca del nome del file da includere dipende da UNIX:

- si cerca in un direttorio specificato dall'utente:
  "nome file"
- in direttori di sistema predefiniti (p.es. /usr/include, /usr/local/include), se il nome del file è specificato tra parentesi angolari: **<nome file>**.

Tipicamente, si può includere il file stdio.h, presente in /usr/include, in questo modo:
#include <stdio.h>



Uso della direttiva include in C

DIRETTIVA #include

Permette di includere un file sorgente (header file) in tutti i file che devono condividere le stesse istruzioni

#include <NomeHeaderFile.h>
#include "NomeHeaderFile.h"

Tutte le istruzioni contenute nel **header file** vengono **inserite** al posto della direttiva e **compilate**

La **forma <...>** specifica un file di inclusione standard (per librerie) - la ricerca avviene in base al path per include fornito al compilatore

La **forma "..."** specifica un file di inclusione di utente - il file viene ricercato prima nel direttorio corrente quindi la ricerca prosegue come nel caso precedente

VISIBILITÀ delle Entità: Dichiarazioni, Definizioni

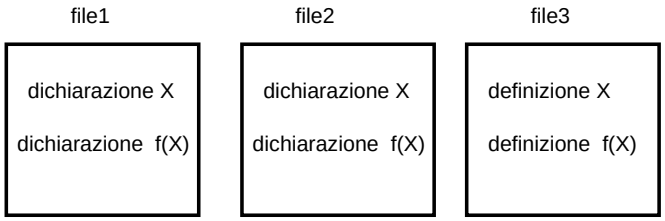
Possibilità di spezzare l'applicazione su **più file**
messi insieme al **collegamento**

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

Esempio: Uno stack

Uno stack che contiene valori reali ===>
data abstraction con funzioni push, pop e isempty, isfull.

La data abstraction stack, in questo caso, è racchiusa tutta in un solo file (*stack.c*) che viene usato dai programmi che ne abbiano necessità

===> **CONDIVISIONE DI DEFINIZIONI**

```
#define DIM 40
#define true 1
#define false 0

static int top = 0;            /* inizializzazione */
static double stack [DIM];
/* strutture dati non visibili al di fuori del file stack.c */
/* le funzioni seguenti sono assunte external a default */
int isempty () { return ((top)? false: true); }
int isfull () { return ((DIM - top)? false: true); }

void push ( f)/* definizione della funzione esportata */
      double f;    /* PROTOTIPO: void push (double f) */
{ stack [top++] = f;  /* non si tratta l'overflow */ }

double pop () {
      double f;
       f = stack [- -top];  /* non si tratta underflow */
       return f;      }
```

Il programma principale, **specificato in un file diverso**, deve dichiarare **extern** le funzioni da **importare**, ma **non può accedere** direttamente alla rappresentazione dello stack (dichiarata **static**)

Ad esempio, nel file *progr.c*:

```
extern int isempty (), isfull ();  /* dichiarazioni per l'uso */
extern void push ();               /* locale delle funzioni   */
extern double pop ();
main () { double a, b; int c;
     do {      printf ("vuoi fare la push: si 1/ no 0\n");
          scanf  ("%d", &c);
          if ((! isfull ()) && (c != 0))
               {    printf (" valore da inserire\n");
                    scanf  ("%le", &a);   push (a); }
          else if (c!=0) puts("stack pieno");
               printf (" vuoi fare la pop: si 1/ no 0\n");
          scanf  ("%d", &c);
          if ((! isempty ()) && (c != 0))
                    {    b = pop ();
                              printf ("\n valore estratto   %e",b); }
               else if (c!=0) puts ("stack vuoto");
               printf (" vuoi uscire: si 1/ no 0\n");
          scanf  ("%d", &c);
        }
     while (c != 1);
}
```

Per ottenere un **unico eseguibile** ===>
bisogna fare il **LINKING** dei file ***stack.c*** e ***progr.c***

Se volessimo mettere il tutto (cioé data abstraction stack e programma main) su **un solo file**, e garantire la **stessa protezione** dovremmo operare in questo modo:

```
#define DIM 40
#define true 1
#define false 0
extern int isempty(), isfull();
extern void push();  /* dichiarazioni per il main */
extern double pop();

main () {
double a, b; int c;
          do {    ...
               }    /* come sopra */
while (c != 1);
}

static int top = 0;             /* inizializzazione */
static double stack [DIM] ;

/*   le   dichiarazioni statiche sono visibili nell'ambito del file di
     definizione solo dopo la definizione stessa (e non prima). Il
     main quindi non può accedervi  direttamente
 */
int isempty () { return ((top) ? false: true); }
int isfull  () { return ((DIM - top)? false: true); }
void push (double f)  { stack [top ++] = f; }
      double pop () { double f; f = stack [top- -];  return f; }
```


Ancora l'esempio dello STACK
utilizzando il concetto di file **HEADER**:

File *stack.h*: **INTERFACCIA DELLO STACK**

```
extern int isempty(), isfull();
extern void push();     /* dichiar. da condividere */
extern double pop();
```

File *stack.c:*

```
#define DIM 40
#define true 1
#define false 0

static int top = 0;             /* inizializzazione */
static double stack [DIM] ;

int isempty () { return ((top) ? false: true); }
int isfull  () { return ((DIM - top)? false: true); }
void push (double f)  { stack [top ++] = f; }
double pop ()
      { double f;   f = stack [top- -];  return f; }
```

File *progr.c:*

```
#include "stack.h"
main () { double a, b; int c;
          do {    ...   }    /* come sopra */
while (c != 1)}
```

Per produrre un **unico eseguibile** ===>
bisogna **LINKARE** progr.c e stack.c **assieme**

Esempio: Una lista

Nel seguito si codifica l'esempio della funzioni relative ad una lista questa volta specificate su **più file**.

Primo file: INTERFACCIA di una LISTA
file ***list.h*** che i programmi per usare le liste devono includere cioé **IMPORTARE**

```
/*  tutte le funzioni seguenti sono dichiarate esterne */
extern void Create (), End (), Enqueue(int i);
extern void EnqueueF(int i), EnqueueL(int i);
extern int  DequeueF (), DequeueL (), Dequeue (int i);
extern int  IsIn (int i), Empty (), Length ();
```

Secondo file: IMPLEMENTAZIONE di una LISTA
file ***list.c*** che contiene l'implementazione di tutte le funzioni di lista

```
#include <stdio.h>
#include <alloc.h>
#define NULL 0

struct node             /* elemento della lista */
     {  int             item;
        struct node    *next;
     };
static struct node *first, *last;
/*puntatori agli elementi iniziale e finale*/
```



```
void Create ()  /* inizializza la coda */
{     first = NULL;  last = NULL;   }

void End ()     /* riazzera la coda  */
{     int    i;
      while (first != NULL) i = DequeueF();
}

int  IsIn (int i)
/* verifica la presenza di un elemento nella coda */
{   struct node *t;
       t = first;
       while (t != NULL && t ->item != i) t = t ->next;
       return(t != NULL);
}

int  Empty ()   /* verifica se la coda è vuota o meno */
{     return (first == NULL);  }

int  Length ()  /* numero degli elementi in lista */
{   int count = 0;
       struct  node *temp = first;
       while (temp != NULL) { count++; temp = temp -> next;}  return
       (count);
}
```

```
void EnqueueF (int i)  /* aggiungi al primo posto in coda */
{   struct node *newnode;
newnode = (struct node *) malloc(sizeof(struct node));
newnode ->next   = first;          newnode ->item   = i;
if (first == NULL) { last = newnode;}
first         = newnode;
}

void EnqueueL (int i)
/* aggiunta all'ultimo posto in coda */
{   struct node *newnode;

      newnode = (struct node *) malloc(sizeof(struct node));
      newnode ->next   = NULL;
      newnode ->item   = i;
      if (first == NULL)
            {    first          = newnode;
            last             = newnode; }
      else
        {     last->next = newnode;
            last        = newnode; }
}

void Enqueue (int i)
/* aggiunta solo se non c'e' in coda */
       {   struct node *t = first;
       while (t != NULL && t -> item != i) t = t -> next;
       if (t == NULL) /* inserisci l'elemento */
              EnqueueF (i);
}
```



```
static Dealloca (struct node *temp)
/* funzione PRIVATA non visibile all'esterno del file*/
{ int reply;
       reply = temp -> item;
       free((char *) temp);
       return reply;
}

int DequeueF  ()    /* togli il primo elemento in coda */
{   struct node *temp = first;
      if (first == NULL) /* coda vuota */ return (NULL);
      else {      if (first == last)
/* un solo elemento da togliere */
                  {     first  = NULL;  last   = NULL; }
            else   first = temp -> next;
            return Dealloca(temp);  }
}

int DequeueL ()   /* togli l'ultimo elemento in coda   */
{   struct node *old, *new, *temp = last;
      if (first == NULL)  /* lista vuota */ return (NULL);
      else { if (first==last)
            /* è il primo elemento da togliere */
            {   last  = NULL; first = NULL; }
           else  /* si lascia almeno un elemento */
            { old = first; new = old -> next;
               while (new != last)
                  { old = new; new = new -> next;}
                last = old;  old ->next = NULL; }
      return Dealloca(temp);
}
```

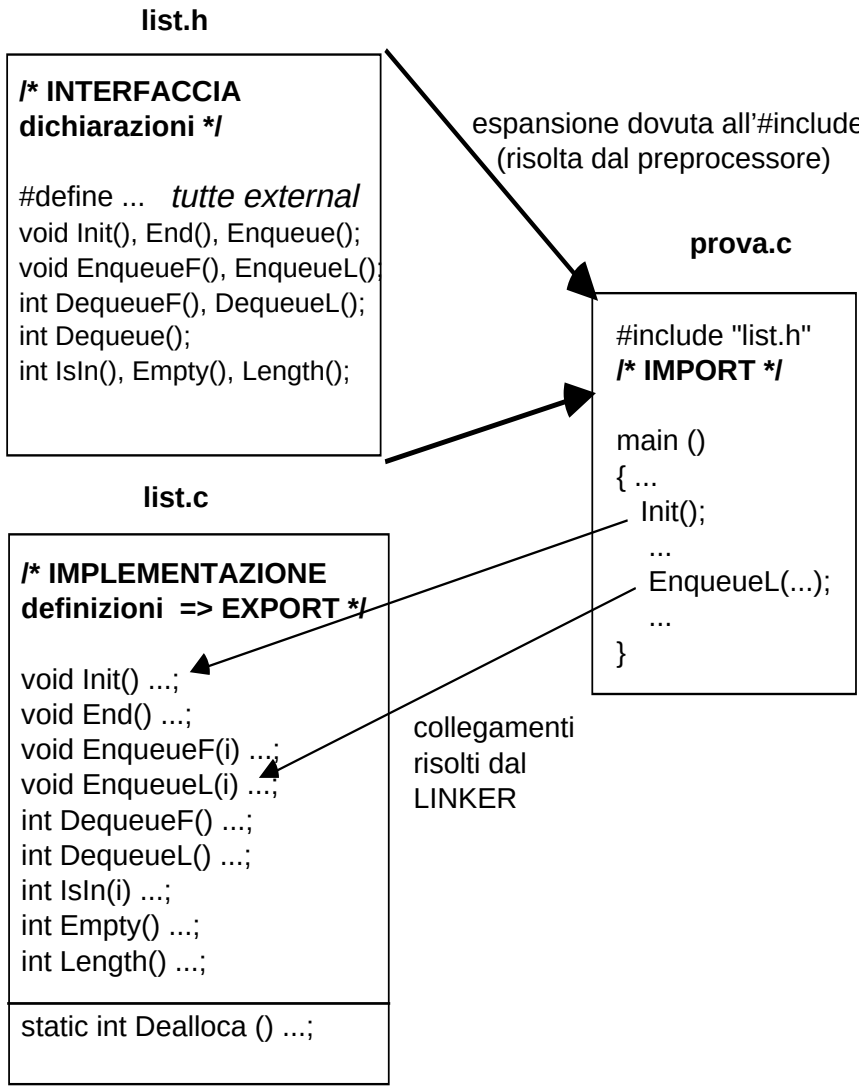
```
int Dequeue (int i)
{      if (first == NULL)  /* lista vuota */ return (NULL);
       else { if (first -> item == i)
                  /* è il primo elemento da togliere*/
                       return Dequeuef();
              else {  struct node *t, *temp;
                           t = first; temp = t -> next;
                       while (temp !=NULL && temp ->item != i)
                                { t = temp; temp = t -> next}
                       if (temp != NULL)  /*l'elemento c'e' */
                                {     if (t -> next == last) last = t;
                                      t -> next = temp -> next;
                                      return Dealloca(temp);
                                }
                       else return NULL;
              /* l'elemento non c'e' */
              }
}
```


ESEMPIO di PROGRAMMA che usa la LISTA:

file *prova.c*:

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

main ()
{ int .....;
      printf
      ("inizio programma di prova della lista su piu' file\n");
      Create();
      EnqueueF(12);
      .....
      DequeueL();
      .....
      End();
}
```

Anche in questo caso per creare un **UNICO ESEGUIBILE** ===> bisogna **COLLEGARE** i file ***prova.c*** e ***list.c*** insieme



MODELLO RUN-TIME DEL C

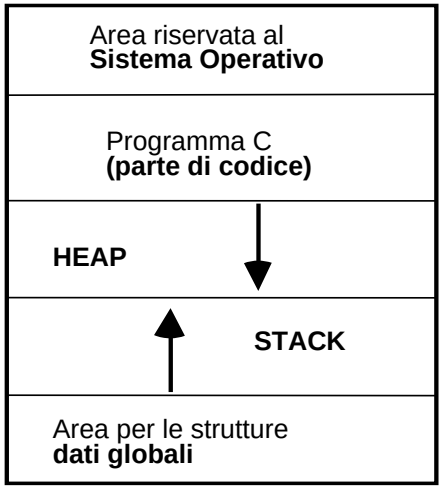
Dobbiamo considerare la *memoria dinamica*:

- **STACK**, per i record di attivazione delle varie procedure (e le variabili automatiche);
- **HEAP**, area da cui allocare le variabili non LIFO. Sono allocate esplicitamente (vedi malloc)

Variabili GLOBALI sono allocate all'inizio, all'atto del caricamento del programma, sia che siano esterne, sia che siano statiche.

L'allocazione può avvenire secondo il seguente schema:

Memoria Utente a RUN-TIME



Heap e Stack crescono
in direzioni opposte

PUNTATORI

puntatori per riferire aree allocate dall'**heap**
ma anche puntatori per accedere ad una qualunque area di memoria

In C non sono forniti strumenti di allocazione di memoria e non sono forniti strumenti di deallocazione, garbage collection della stessa ==>

Affidato a primitive di UNIX proprie del Sistema Operativo come ad esempio:

char ***malloc*** (unsigned size);
La funzione malloc alloca uno spazio di memoria heap di ampiezza size

void **free** (char *ptr);
La funzione free libera l'area puntata da ptr.

Un esempio di questo è anche l'intero insieme
delle funzioni di Input/Output

Si veda il *passaggio dei parametri* per il C
ed il *modello di memoria* a basso livello

HEAP

Un **heap** è una lista dei ***blocchi di memoria liberi*** disponibili da cui operare con allocazioni ed a cui aggiungere ***blocchi liberati***

Ogni *malloc* ricerca un blocco di dimensioni sufficienti a contenere la richiesta (*malloc(size)*)
Ogni *free* deve potere restituire il blocco di memoria allocato precedentemente

Necessità di separare politiche dai meccanismi

meccanismo: ogni blocco allocato ha un header che ne specifica la lunghezza, invisibile all'utente

politiche:

Allocazione

effettuare una scansione completa della lista?
quale blocco scegliere (se più di uno disponibile)?

- **first fit** primo blocco trovato
- **best fit** migliore blocco trovato (cioè dimensione più vicina alla richiesta)
- **worst fit** peggiore approssimazione alla richiesta; vantaggio di lasciare blocchi liberi consistenti in dimensione

Deallocazione

come si aggancia alla lista dei blocchi liberi
per esempio tenuta in ordine di indirizzo

Garbage collector

GC programma che trova i blocchi di memoria (di un *heap*, ad esempio) che non sono più riferibili e li rende disponibili per un successivo utilizzo

Un ***GC*** deve provvedere a verificare quali parti sono ancora in uso e quali non lo sono: la lora memoria può essere messa a disposizione di altre allocazioni

In un *linguaggio di programmazione*, quali parti di heap possono essere ritrovate da almeno un *puntatore* e quali *non* siano recuperabili

quelle non recuperabili sono *liberate* **di ufficio** e ricollegate all'heap

Complessità di un GC

è necessario scandire l'intero programma applicativo, mentre il programma esegue, e seguire le catene dei puntatori per ottenere le aree ancora riferibili

E se più programmi usano lo stesso heap?

Compattazione

fase di riorganizzazione della memoria per fondere **più blocchi** liberi in un **unico blocco** di dimensioni superiori

Problema

una compattazione globale richiede di variare anche indirizzi di aree già allocate che devono essere modificate in modo trasparente a chi le usa

Solo lo **spostamento** di aree usate consente di compattare le aree libere

==>

è necessario conoscere tutti i **puntatori** ad ogni area che deve essere spostata

per modificarne il contenuto

Soluzione

una compattazione è fattibile solo in sistemi in cui ogni **puntatore** non sia un *riferimento diretto* alla memoria,

ma l'accesso avvenga attraverso tabelle mantenute dal sistema operativo/hardware

(vedi memoria virtuale)