



di.unito.it

DIPARTIMENTO
DI INFORMATICA

DI INFORMATICA
DIPARTIMENTO

di.unito.it

laboratorio di sistemi
operativi — UNIX

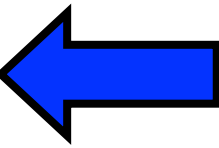
complementi di C

Marco Botta

Materiale a cura di Daniele Radicioni

argomenti del laboratorio UNIX

1. introduzione a UNIX;
2. integrazione C: precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. segnali;
4. controllo dei processi;
5. pipe e fifo;
6. code di messaggi;
7. memoria condivisa;
8. semafori;
9. introduzione alla programmazione bash.



programma

- operatori bitwise e precedenze fra gli operatori
- classi di memorizzazione
- ripasso puntatori
- il preprocessore C
- la pacchettizzazione del codice
 - compilazione condizionale
 - l'utility make
- esercizi

bitwise operators

bitwise operators

- il C permette ai programmatori di sistemi di **manipolare i bit**; questo insieme di operazioni prima dell'avvento del C era stato la provincia del programmatore di **assembly**, e il codice risultante era per definizione **non-portabile**
- **bit-twiddling** è la manipolazione dei singoli bit in variabili intere
 - nessuno degli operatori bitwise può essere utilizzato su operandi di tipo ***real***, perché il tipo ***real*** non consente l'accesso ai singoli bit

the six bitwise operators

Operator	Effect	Conversions
&	bitwise AND	usual arithmetic conversions
	bitwise OR	usual arithmetic conversions
^	Bitwise XOR	usual arithmetic conversions
<<	left shift	integral promotions
>>	right shift	integral promotions
~	one's complement	integral promotions

```
int x=86, y=50;
printf("x&y = %d\n", x&y);
```

```

  0 1 0 1 0 1 1 0
& 0 0 1 1 0 0 1 0
-----
  0 0 0 1 0 0 1 0
```

```
int x=86, y=50;
printf("x|y = %d\n", x|y);
```

```

  0 1 0 1 0 1 1 0
| 0 0 1 1 0 0 1 0
-----
  0 1 1 1 0 1 1 0
```

```
int x=86, y=50;
printf("x^y = %d\n", x^y);
```

```

  0 1 0 1 0 1 1 0
^ 0 0 1 1 0 0 1 0
-----
  0 1 1 0 0 1 0 0
```

```
int x=86;  
printf("x<<2 = %d\n", x<<2);
```

```
    0 1 0 1 0 1 1 0 << 2  
-----  
0 1 0 1 0 1 1 0 0 0
```

```
int x=86;  
printf("x >> 1 = %d\n", x>>1);
```

```
    0 1 0 1 0 1 1 0 >> 1  
-----  
    0 1 0 1 0 1 1
```


esercizio

- come possiamo utilizzare gli operatori *bit-a-bit* per determinare se un numero è pari o dispari ?

soluzione

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

int main(void) {
    int i;

    for(i=1; i <= 10; i++)
        printf((i&1) ? "%2d odd\n" : "%2d even\n", i);

    exit(EXIT_SUCCESS);
}
```

```
...  
  
for (i=1; i <= 10; ++i)  
    printf( (i&1) ? "%2d odd\n" : "%2d even\n", i );  
  
...
```

```
bash-3.2$ ./odd-and-even
```

```
1 odd  
2 even  
3 odd  
4 even  
5 odd  
6 even  
7 odd  
8 even  
9 odd  
10 even
```



precedenze...

Operator	Direction
() [] -> .	left to right
! ~ ++ -- - + (cast) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += and all combinations	right to left
,	left to right

un aiuto per ricordare la tabella:

- gli operatori con precedenza massima sono le parentesi delle funzioni, l'operatore per gli indici e gli operatori di accesso ai membri;
- seguono gli operatori unari;
- seguiti a loro volta da quelli aritmetici (operatori moltiplicativi hanno priorità più alta di quelli additivi);
- tutti i tipi di assegnamento hanno priorità più bassa, tranne che rispetto all'operatore virgola, la cui priorità è minima.

precedenze

- come si fa a riconoscere un operatore binario
 1. $++$ e $--$ sono sempre operatori *unari*
 2. l'operatore immediatamente a destra di un operando è un operatore *binario*
 3. tutti gli operatori alla sinistra di un operando sono *unari*

precedenze

- gli operatori *unari* hanno *alta precedenza*; quindi l'espressione

$a + \underline{-b++} + c$

precedenze

- gli operatori *unari* hanno *alta precedenza*; quindi l'espressione

$a + \underline{-b++} + c$

- ha due operatori *unari* applicati a *b*. poiché tutti *gli operatori unari associano da destra a sinistra (R L)*, benché il segno '-' venga prima dell'operatore di incremento quando si legge, la parentesizzazione corretta è:

$a + -(b++) + c$

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

?

?

?

precedenze

- gli operatori *unari* hanno *alta precedenza*; quindi l'espressione

$a + -b++ + c$

- ha due operatori *unari* applicati a *b*. poiché tutti *gli operatori unari associano da destra a sinistra (R L)*, benché il segno - venga prima dell'operatore di incremento quando si legge, la parentesizzazione corretta è:

$a + -(b++) + c$

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

```
a      = 5
-a++   = -5
a      = 6
```

associatività

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

main() {
    int i,j;
    float f;

    i = 5; j = 2;
    f = 3.0;

    f = f + j / i;
    printf("value of f is %f\n", f);
    exit(EXIT_SUCCESS);
}
```

value of f is ???

associatività

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

main() {
    int i,j;
    float f;

    i = 5; j = 2;
    f = 3.0;

    f = f + j / i;
    printf("value of f is %f\n", f);
    exit(EXIT_SUCCESS);
}
```

value of f is **3.000000**

- avremmo potuto pensare che poiché c'era un **float** coinvolto nell'espressione, l'intera espressione (divisione inclusa) sarebbe stata eseguita in quel tipo

associatività

1

poiché l'operatore della divisione aveva tipi *int* da entrambe le parti, l'operazione aritmetica eseguita è una **divisione intera**, con risultato zero

3

la somma di un *float* e un *int* richiede di convertire l'intero in *float* per eseguire l'operazione

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

main() {
    int i, j;
    float f;

    i = 5; j = 2;
    f = 3.0;

    f = f + j / i;
    printf("value of f is %f\n", f);
    exit(EXIT_SUCCESS);
}
```

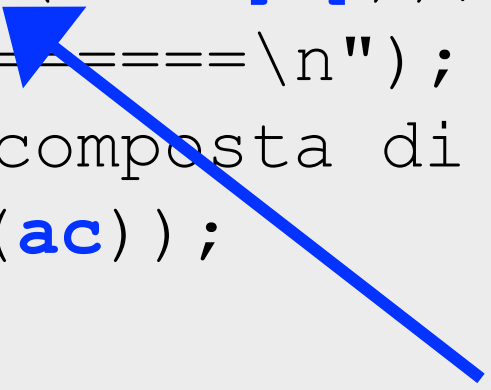
poiché quello è il tipo corretto per l'assegnamento, non ci sono ulteriori conversioni

2

```
...
#define ARSZ 10

int main(void) {
    int i;
    char ac[ARSZ] =
        {'a','b','c','d','e','f','g','h','i','\0'};
    char arc[] = "ciao";
    char *ap;

    for(i=0; i<ARSZ; ++i)
        printf("ac[%d]: %c\n", i, ac[i]);
    for(ap=ac; ap<&ac[ARSZ-1]; ++ap)
        printf("ap value: %c \t address: %p\n", *ap, ap);
    printf("=====\n");
    printf("la stringa %s e` composta di %d caratteri\n",
        arc, strlen_ridef(&arc[0]));
    printf("=====\n");
    printf("la stringa %s e` composta di %d caratteri\n",
        ac, strlen_ridef(ac));
    return 0;
}
```



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

```
#define ARSZ 10
```

```
int main(void) {
```

```
    int i;
```

```
    char ac[ARSZ]={'a','b','c','d','e','f','g','h','i','\0'};
```

```
    char arc[] = "ciao";
```

```
    char *ap;
```

```
    for(i=0; i<ARSZ; ++i)
```

```
        printf("ac[%d]: %c\n", i, ac[i]);
```

```
    for(ap=ac; ap<&ac[ARSZ-1]; ++ap)
```

```
        printf("ap value: %c \t address: %p\n", *ap, ap);
```

```
    printf("=====\n");
```

```
    printf("la stringa %s e` composta di %d caratteri\n",
```

```
        arc, strlen_ridef(&arc[0]));
```

```
    printf("=====\n");
```

```
    printf("la stringa %s e` composta di %d caratteri\n",
```

```
        ac, strlen_ridef(ac));
```

```
    return 0;
```

```
}
```

```
int strlen_ridef(char *cp) {
    char *cursore = cp;
    while(*cursore++);
    return cursore-1-cp;
}
```

classi di memorizzazione

classi di memorizzazione

- tutte le variabili e le funzioni del C hanno 2 attributi: la *classe di memorizzazione* e il *tipo*
- le 4 classi di memorizzazione possibili sono le seguenti: *auto*, *extern*, *register*, *static*

visibilità *auto*

- le variabili che compaiono nel *main* sono dette *private* o *locali al main*: le altre funzioni non possono accedervi direttamente
 - lo stesso vale per le variabili delle altre funzioni, che si ‘materializzano’ solo al momento dell’invocazione della funzione, per svanire al termine
- poiché le variabili automatiche *appaiono e scompaiono* in conseguenza delle chiamate alle funzioni, non possono mantenere i loro valori fra una chiamata e l’altra, e devono essere esplicitamente *inizializzate* a ogni chiamata;
 - diversamente, conterranno valori ‘sporchi’

visibilità *auto*

- poiché la classe di memorizzazione di default è *auto*, la parola chiave *auto* si usa di rado
- all'ingresso in un blocco il sistema *alloca* memoria per le *variabili automatiche*; pertanto tali variabili sono considerate locali al blocco
- all'uscita dal blocco, il sistema *libera la memoria* assegnata alle variabili automatiche, causando la perdita dei loro valori
- al rientro nel blocco, il sistema *alloca nuovamente la memoria* senza però recuperare i valori precedenti

extern

- è anche possibile definire variabili **esterne** a tutte le funzioni, ovvero accessibili da parte di qualsiasi funzione
- vista la loro **accessibilità globale**, le variabili esterne possono sostituire le liste di argomenti per lo scambio di dati fra le funzioni; **mantengono i loro valori** anche dopo che le funzioni che le hanno modificate hanno smesso di operare
- una variabile **extern** si **definisce** una sola volta al di fuori di qualunque funzione; la definizione porta il sistema a riservare una quantità appropriata di memoria per i contenuti della variabile
- la variabile deve anche essere **dichiarata in ogni funzione che intenda accedervi**; la dichiarazione può essere un'istruzione **extern** o risultare implicitamente dal contesto

extern

```
#include <stdio.h>

int    a = 1, b = 2, c = 3;
int    f(void);

int main(void) {
    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);
    return 0;
}
```

```
int f(void) {
    extern int    a;
    int          b, c;

    a = b = c = 4;
    return (a + b + c);
}
```

variabili **globali** rispetto a tutte le funzioni dichiarate dopo; continuano a esistere all'uscita dal blocco o dalla funzione

essendo dichiarate al di fuori di una funzione hanno classe di memorizzazione **extern**, anche se la parola chiave **extern** non è specificata

il meccanismo delle variabili esterne costituisce un **modo per passare le informazioni**. tuttavia per aumentare la modularità del codice e per ridurre gli effetti collaterali, **è preferibile utilizzare il passaggio dei parametri**



```

#include <stdio.h>

int    a = 1, b = 2, c = 3;
int    f(void);

int main(void) {
    printf("%3d\n", f());
    printf("%3d%3d%3d\n", a, b, c);
    return 0;
}

int f(void) {
    extern int    a;
    int          b, c;

    a = b = c = 4;
    return (a + b + c);
}

```

```
$ ./mio_prog
```

12

4 2 3



register

- classe di memorizzazione che ha come obiettivo l'aumento della **velocità di esecuzione**
- segnala al compilatore che la variabile corrispondente dovrebbe essere memorizzata in **registri di memoria ad alta velocità**

qualora il compilatore non possa allocare un registro fisico, viene utilizzata la classe **automatica** (il compilatore dispone solo di una parte dei registri, che possono invece essere utilizzati dal sistema)

- quando la velocità è importante, il programmatore può scegliere **poche variabili** alle quali viene fatto più frequentemente accesso (per esempio, variabili di ciclo e parametri delle funzioni)

```
register int i;  
for(i = 0; i < LIMIT; ++i)  
    ...
```

register

- all'uscita dal blocco, il registro viene liberato
- *register int i;* equivale a *register i;*
- la variabile *register* è di norma dichiarata nel punto più vicino possibile al punto in cui viene utilizzata, per consentire la massima disponibilità di registri fisici, utilizzati solo quando necessario

classe *static*

- variabili *static* servono per permettere a una variabile locale di mantenere il valore precedente al rientro in un blocco

```
void f(void) {  
    static int count = 0;  
    ++count;  
    ...  
}
```

alla prima chiamata della funzione **count** viene inizializzata a zero;

alle chiamate successive **non viene più inizializzata**, ma mantiene il valore che aveva alla precedente chiamata di funzione

variabili *static extern*

- questo tipo di classe di memorizzazione fornisce meccanismo di ‘privatezza’ (insieme di restrizioni sulla visibilità di variabili o funzioni che sarebbero altrimenti accessibili) fondamentale per la modularità dei programmi
- le variabili *statiche esterne* sono variabili *esterne* con visibilità ristretta: sono accessibili dal resto del file in cui sono dichiarate
 - non sono disponibili alle funzioni precedentemente definite all'interno del file o definite all'interno di file differenti, anche se tali funzioni utilizzano la parola chiave *extern* relativamente alla classe di memorizzazione

esempio

```
void f(void) {  
    ... // v non disponibile  
}
```

```
static int v;
```

```
void g(void) {  
    ... // v disponibile  
}
```

- l'idea è cioè di disporre di una variabile globale per una famiglia di funzioni, e al contempo privata per il file

inizializzazione di default

- sia le variabili **statiche** sia quelle **esterne** non inizializzate esplicitamente vengono **inizializzate a zero** dal sistema
 - anche array, stringhe, puntatori, strutture e unioni subiscono lo stesso trattamento: per array e stringhe ogni elemento viene posto a zero
- al contrario, normalmente le variabili **automatiche** e **registro non** vengono **inizializzate** dal sistema, e quindi contengono inizialmente valori ***sporchi***

ripasso sui puntatori

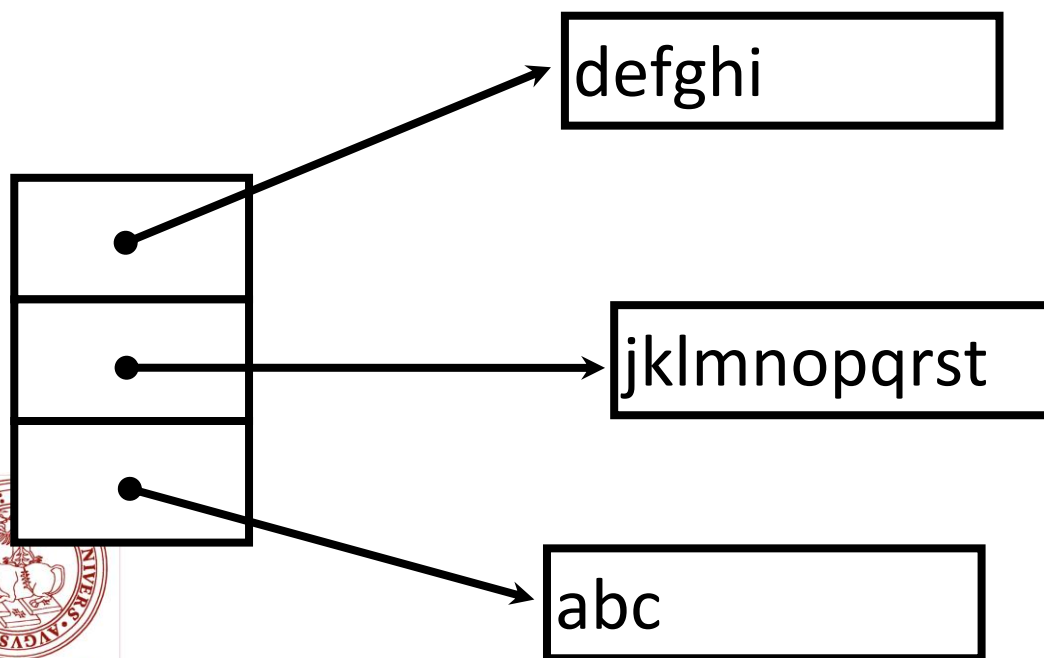
array di puntatori

array di puntatori

- i puntatori possono essere memorizzati in array, poiché sono essi stessi delle variabili;
- immaginiamo di volere ordinare delle righe di testo di lunghezza variabile.
- se le righe di testo da ordinare sono memorizzate consecutivamente in un lungo vettore di caratteri, è possibile accedere a ogni riga tramite un **puntatore al primo carattere**.
- gli stessi puntatori possono essere memorizzati in un vettore.

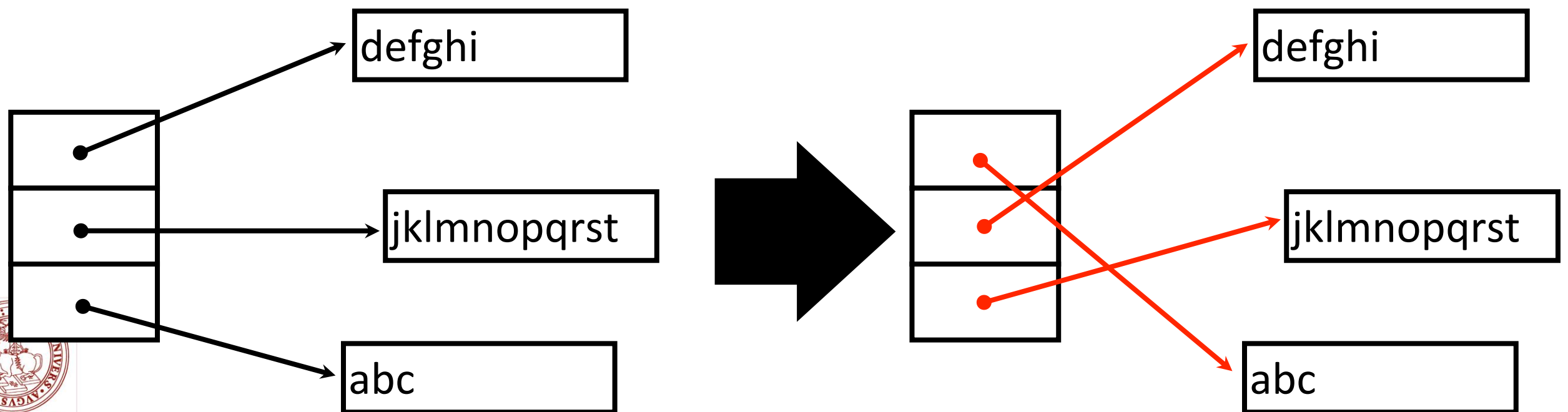
array di puntatori (2)

- per **confrontare due righe** si potrebbero passare i puntatori a ***strcmp***;
- quando due righe sono fuori posto e devono essere scambiate, sono i puntatori a mutare collocazione nel vettore di puntatori, e non le righe stesse...

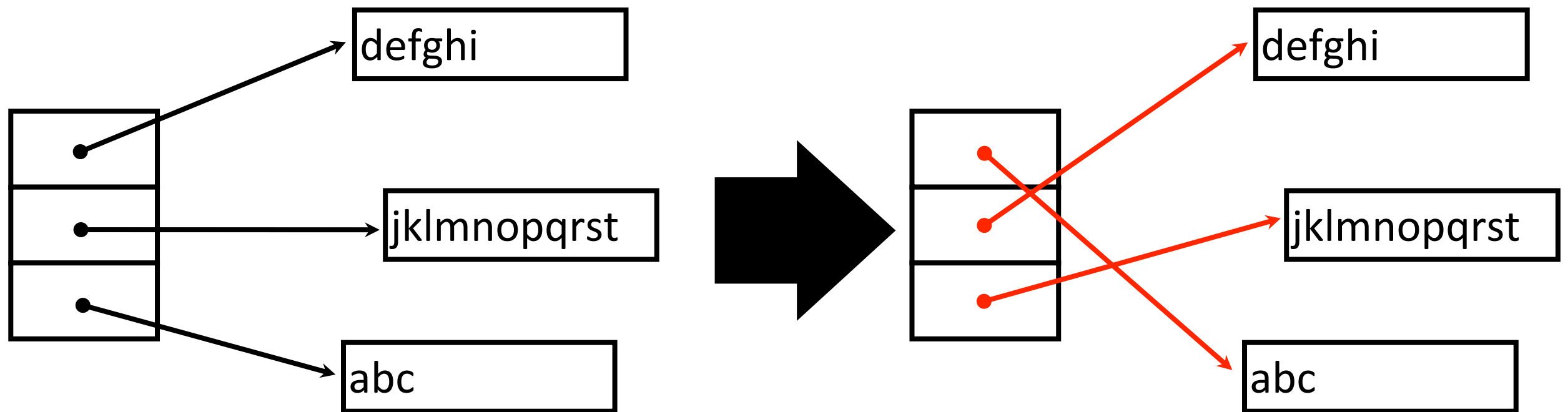


array di puntatori (2)

- per **confrontare due righe** si potrebbero passare i puntatori a ***strcmp***;
- quando due righe sono fuori posto e devono essere scambiate, sono i puntatori a mutare collocazione nel vettore di puntatori, e non le stringhe stesse...



array di puntatori (3)



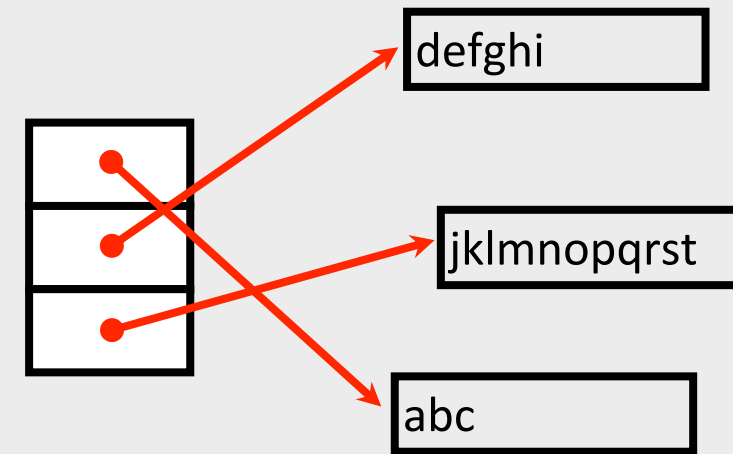
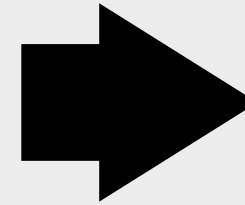
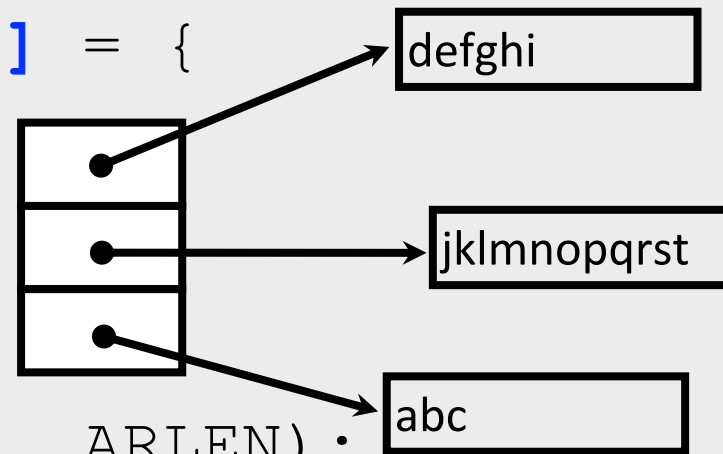
- otteniamo così un doppio vantaggio legato allo spostamento fisico delle stringhe:
 - la gestione della memoria; e
 - la penalizzazione delle prestazioni.

```
#define ARLEN 3
void ordina(char* alfabeto[], int length);
```

```
main() {
```

```
    char* array_char_p[] = {
        "defghi",
        "jklmnopqrst",
        "abc"
    };
```

```
    ordina(array_char_p, ARLEN);
```



```
void ordina(char* array_char_p[], int length) {
    int i, j;    char* tmp;
    for( i=0; i< length; i++ ) {
        for(j=i+1; j< length; j++) {
            if( strcmp(array_char_p[i], array_char_p[j]) > 0 ) {
                tmp = array_char_p[i];
                array_char_p[i] = array_char_p[j];
                array_char_p[j] = tmp;
            }
        }
    }
}
```

argomenti dalla riga di comando

```
int main(int argc, char *argv[]) {  
    int i;  
    printf("argc = %d\n", argc);  
    for(i=0; i<argc; ++i)  
        printf("argv[%d] = %s\n", i, argv[i]);  
    return 0;  
}
```

```
$ ./lavagna primo secondo terzo quarto quinto
```

```
argc = 6
```

```
argv[0] = ./lavagna
```

```
argv[1] = primo
```

```
argv[2] = secondo
```

```
argv[3] = terzo
```

```
argv[4] = quarto
```

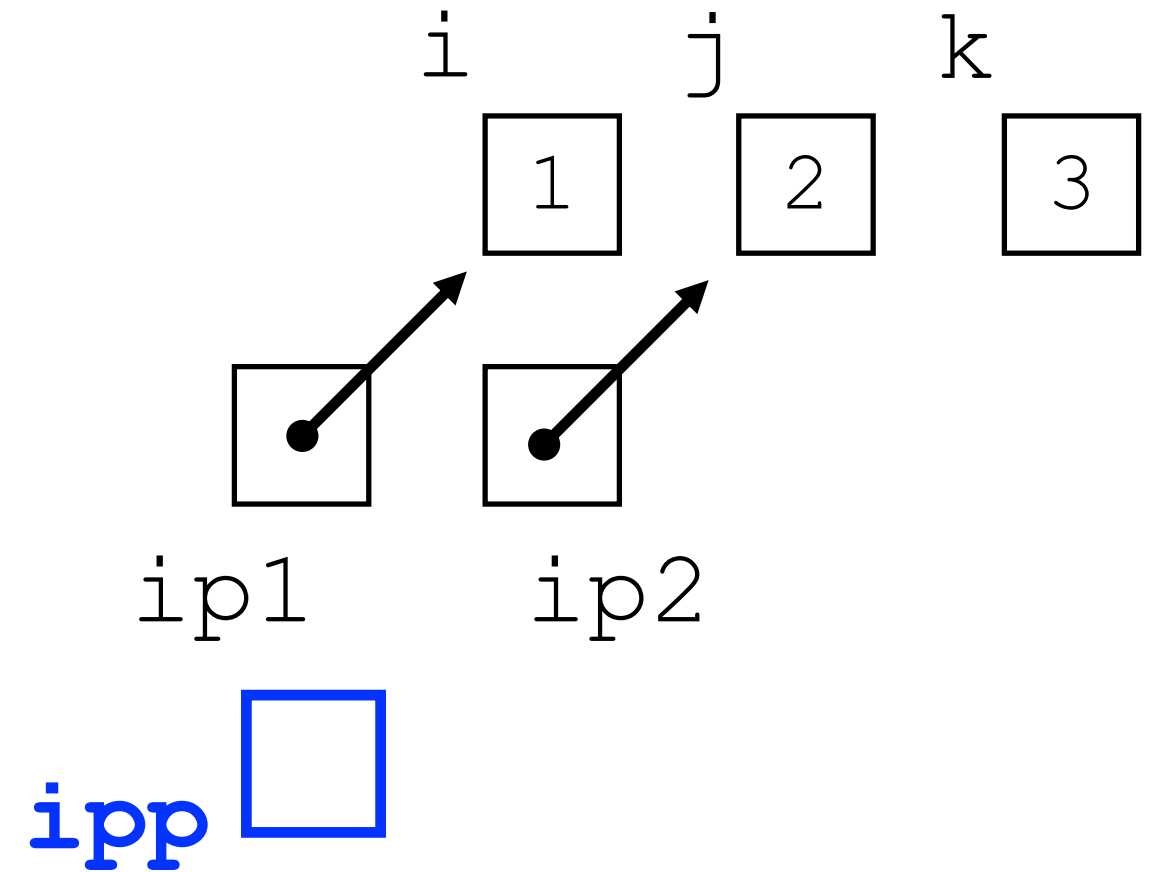
```
argv[5] = quinto
```

argc contiene il numero dei parametri sulla riga di comando

argv è un **array di puntatori a char**: un array di stringhe, ciascuna delle quali è una parola sulla riga di comando

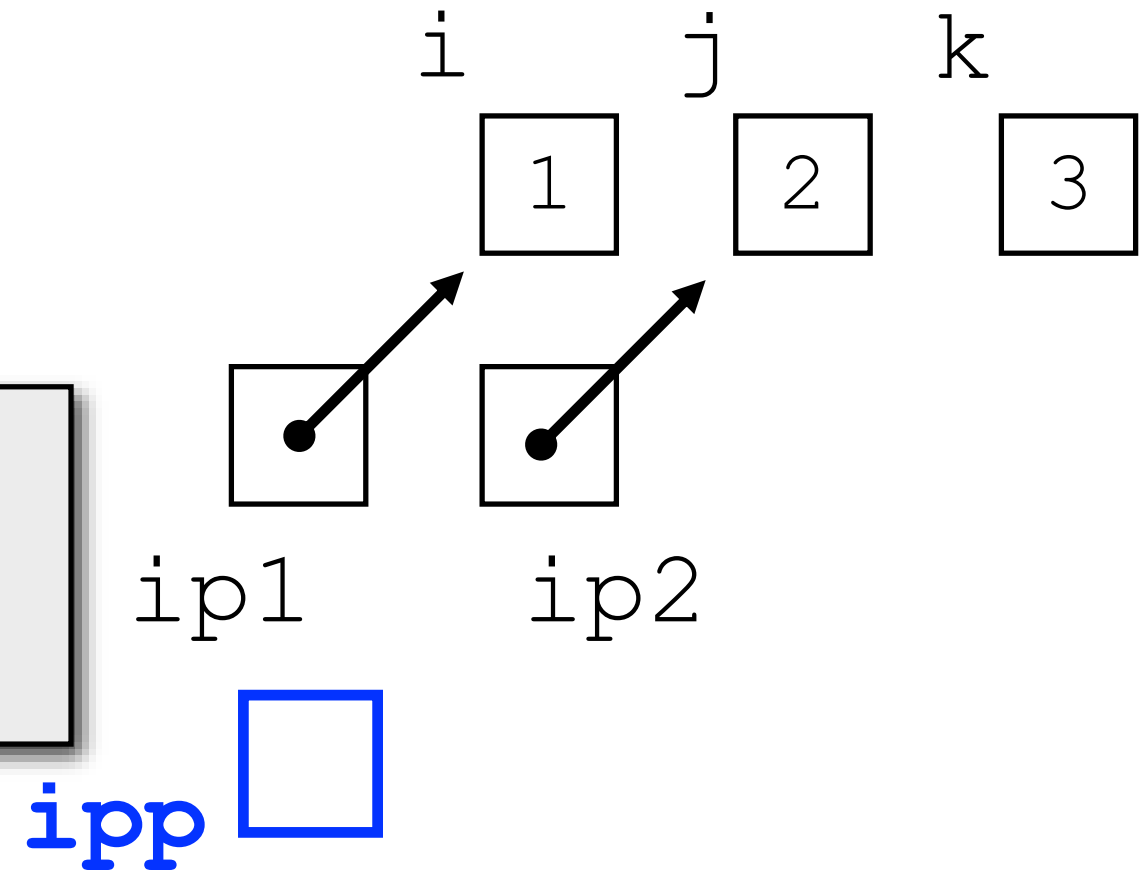
indirezione multipla

indirizzazione multipla



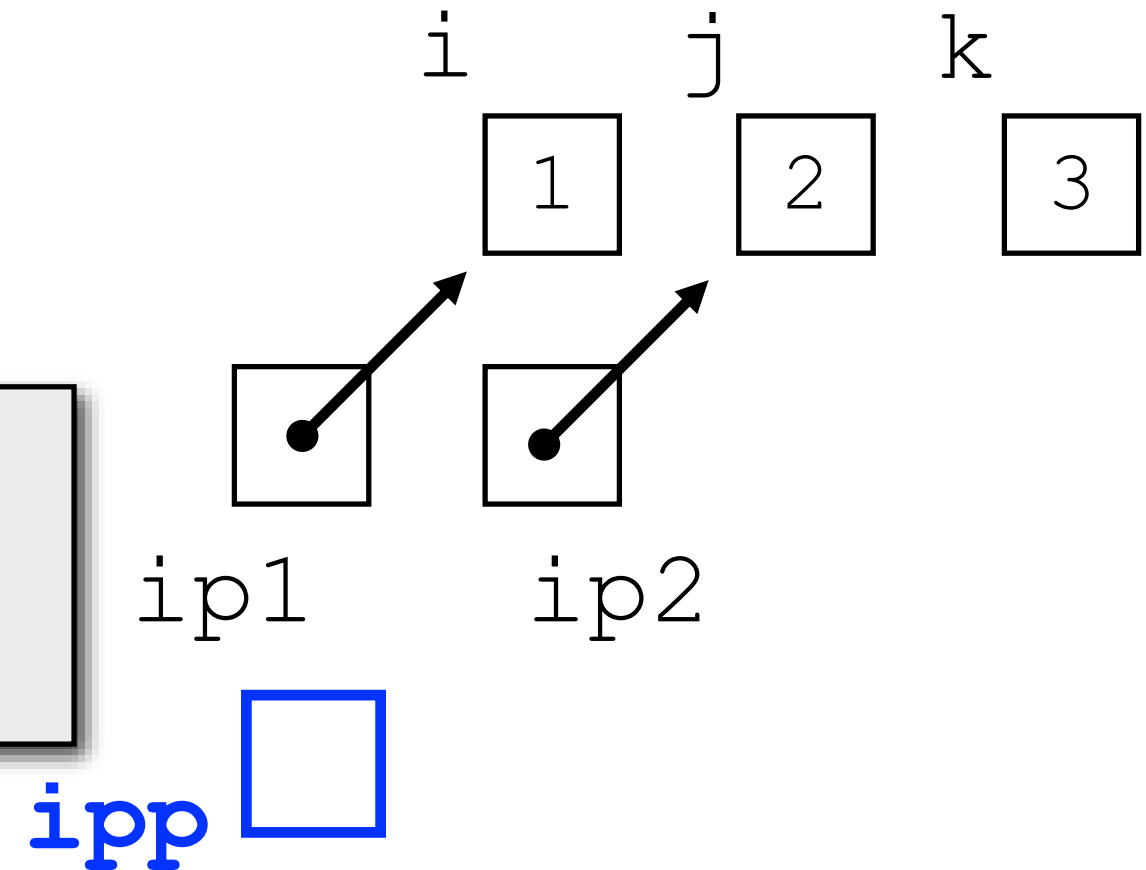
indirizzazione multipla

```
int **ipp;  
int i = 1, j = 2; k = 3;  
int *ip1 = &i, *ip2 = &j;
```

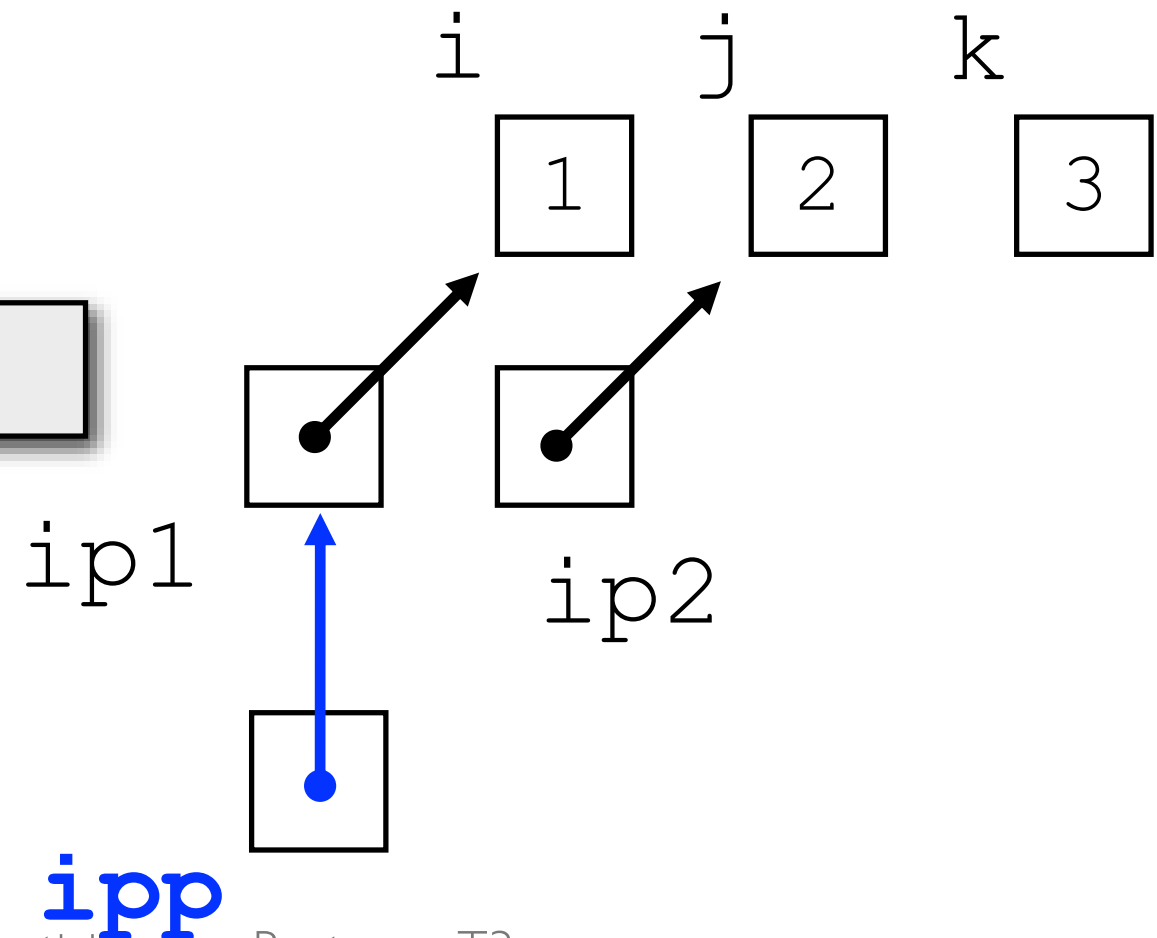


indirizzazione multipla

```
int **ipp;  
int i = 1, j = 2; k = 3;  
int *ip1 = &i, *ip2 = &j;
```



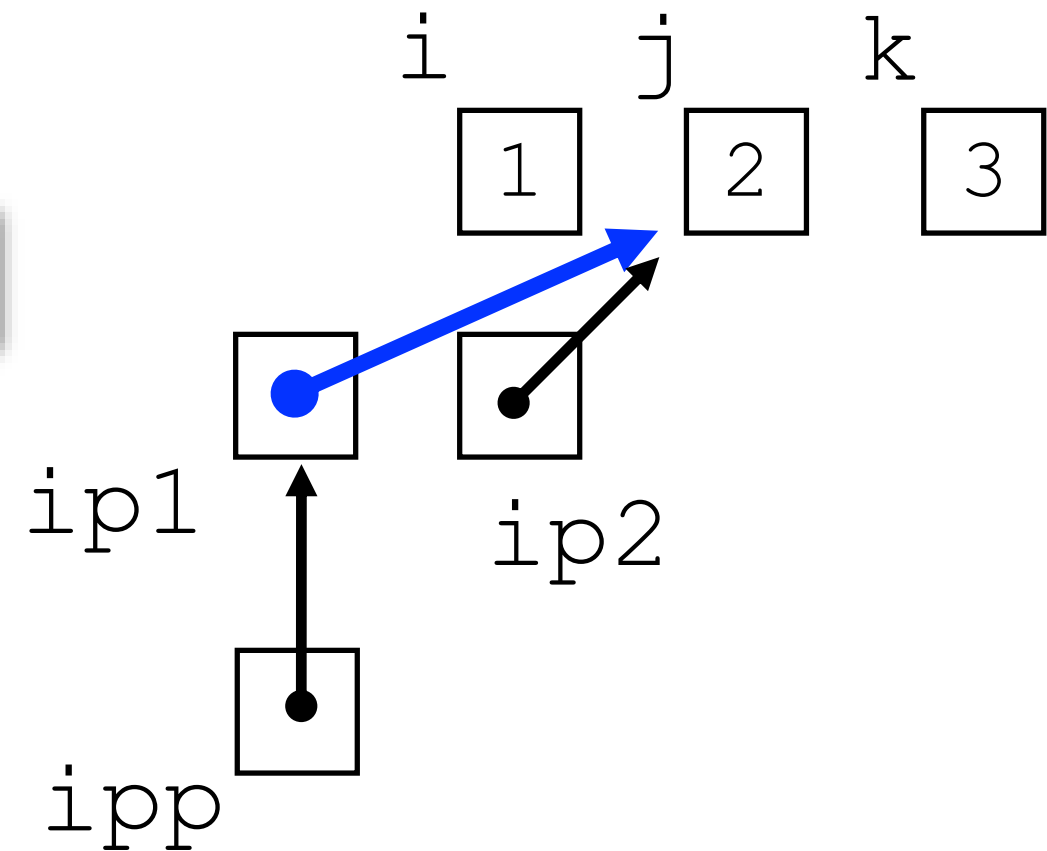
```
ipp = &ip1;
```



indirizzazione multipla

```
*ipp = ip2;
```

- dereferenziazione di *ipp*:
otteniamo *ip1*
- assegnamento a *ip1* del
contenuto di *ip2*: *ipp* punta
a *j*

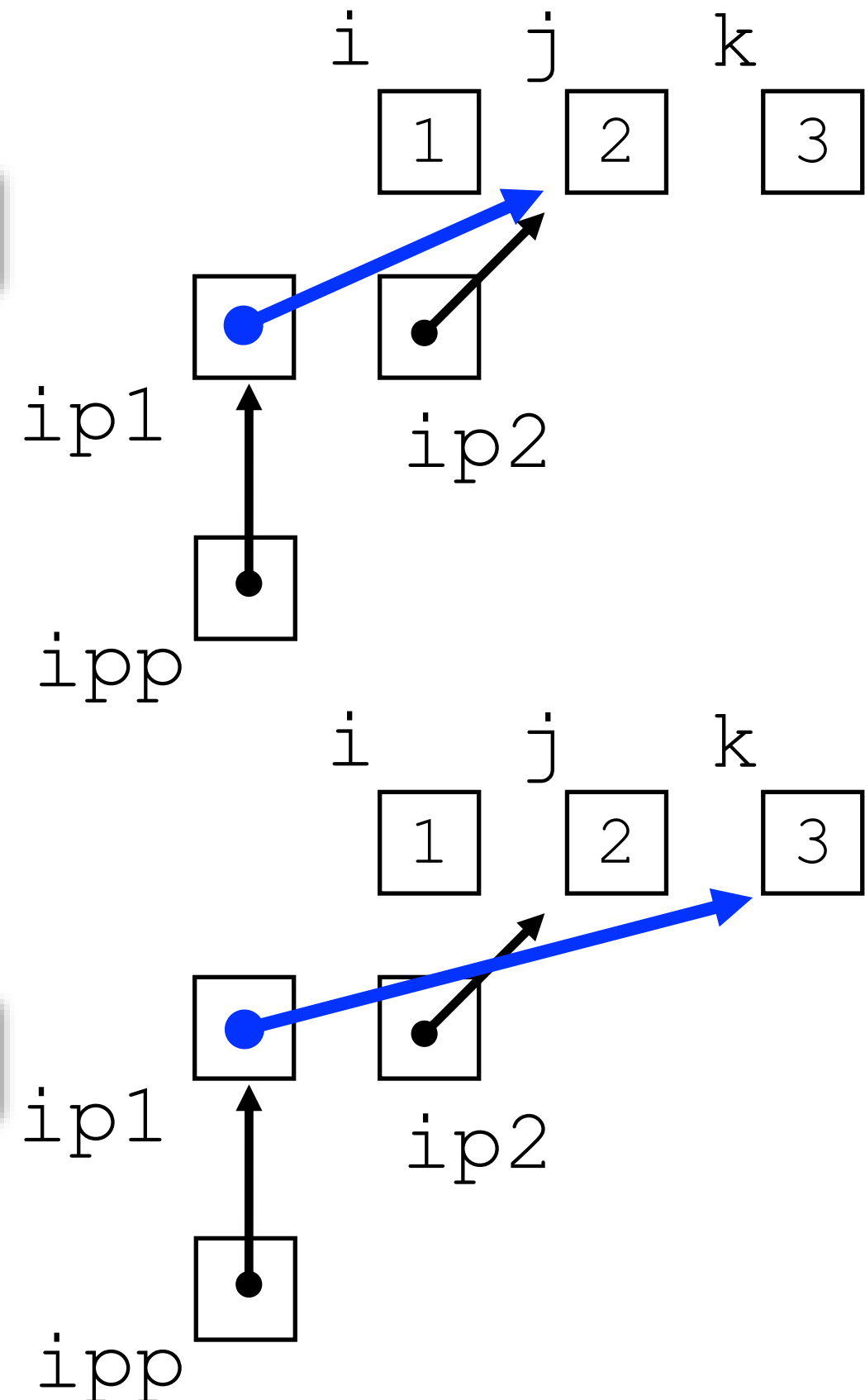


indirizzazione multipla

```
*ipp = ip2;
```

- dereferenziazione di *ipp*:
otteniamo *ip1*
- assegnamento a *ip1* del
contenuto di *ip2*: *ipp* punta
a *j*

```
*ipp = &k;
```



dall'esempio dell'array di puntatori...

```
#define ARLEN 3

void ordina(char* array_char_p[], int length) {
    int i, j;    char* tmp;
    for(i=0; i< length; ++i)
        for(j=i+1; j< length; ++j)
            if(strcmp(array_char_p[i], array_char_p[j])>0) {
                swap(){ tmp = array_char_p[i];
array_char_p[i] = array_char_p[j];
array_char_p[j] = tmp;
}
            }
}
```

- vorremmo costruire una funzione *swap()* che ci permetta di scambiare i puntatori contenuti nell'array

```

#define ARLEN 3

void ordina(char* array_char_p[], int length) {
    int i, j;    char* tmp;
    for(i=0; i< length; ++i)
        for(j=i+1; j< length; ++j)
            if(strcmp(array_char_p[i], array_char_p[j])>0) {
                swap_strings(????,????) ;
            }
}

```

```

int swap_strings(????,????) {
    . . .
}

```

indirizzazione multipla

- il chiamante utilizza la funzione *swap* con l'istruzione
swap(&array[i], &array[j]);
- ogni argomento di *swap()* è un indirizzo di un puntatore a *char*, o equivalentemente un puntatore a puntatore a *char*
- per questo motivo i parametri formali nella definizione sono di tipo *char ***.

```
void swap(char **p, char **q) {  
    char *tmp;  
    tmp = *p;  
    *p = *q;  
    *q = tmp;  
}
```

poiché p è un puntatore a puntatore a char, l'espressione *p che dereferenzia p è di tipo puntatore a char, cioè *char**

```
#define ARLEN 3

void ordina(char* array_char_p[], int length) {
    int i, j;    char* tmp;
    for(i=0; i< length; ++i)
        for(j=i+1; j< length; ++j)
            if(strcmp(array_char_p[i], array_char_p[j])>0) {
                swap_strings(&array_char_p[i], &array_char_p[j]);
            }
}
```

```
void swap_strings(char **prima, char **seconda) {
    char *tmp;
    tmp = *prima;
    *prima = *seconda;
    *seconda = tmp;
}
```

puntatori a funzioni

pointers to functions

```
int func(int a, float b);
```

- their **declaration** is easy: write the declaration as it would be for the function.
- and simply **put brackets around the name and a *** in front of it: that declares the pointer. because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int */  
int *func(int a, float b);  
/* pointer to function returning int */  
int (*func)(int a, float b);
```

usage

- once you've got the pointer, you can assign the address of the right sort of function just by using its name;
- like an array, a function name is turned into an address when it's used in an expression. you can call the function using one of two forms:

```
(*func) (1, 2) ;  
func (1, 2) ;
```


usage example

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

void func(int);

int main(void) {
    void (*fp) (int) ;
    fp = func;
    (*fp) (1) ;
    fp (2) ;
    return 0;
}

void func(int arg) {
    printf("%d\n", arg);
}
```

preprocessore

preprocessore

- le righe che iniziano con il simbolo **#** sono dette direttive al preprocessore
- una direttiva al preprocessore ha effetto a partire dal punto in cui si trova
 - fino alla fine del file stesso, o
 - fino al raggiungimento di un'altra direttiva che ne neghi l'effetto

preprocessore

- le direttive al preprocessore possono contenere **espressioni complesse**, come costrutti *if-then-else*
- usando le direttive possiamo scrivere meta-programmi
- queste istruzioni sono **interpretate** e rimosse dal preprocessore **prima che il codice arrivi al compilatore**

alterazioni del linguaggio

```
PROGRAM  
BEGIN  
    WriteLn(ESEMPIO_DI_USO DELLE_MACRO) ;  
END.
```

alterazioni del linguaggio

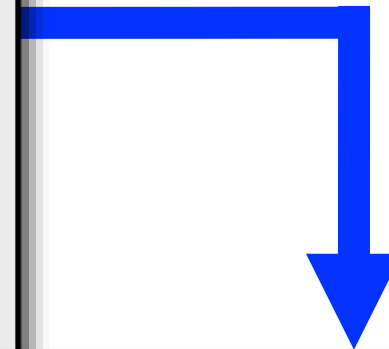
```
#include <stdio.h>

#define PROGRAM int
#define BEGIN main() {
#define END }
#define ESEMPIO_DI_USO DELLE_MACRO \
    "\n funziona??? \n"

#define WriteLn printf

PROGRAM
BEGIN
    WriteLn(ESEMPIO_DI_USO DELLE_MACRO);
END;
```

le direttive costituiscono uno strumento molto potente per [alterare il linguaggio](#)



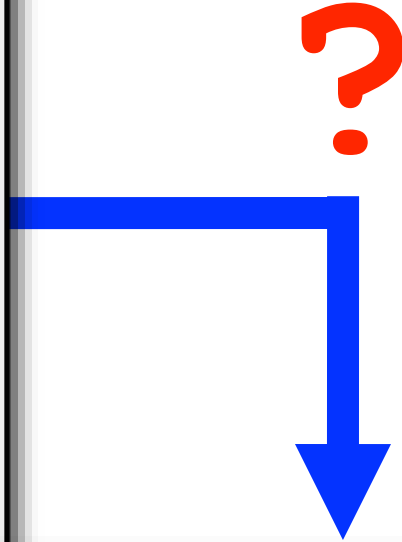
```
#include <stdio.h>

int
main() {
    printf("\n funziona??? \n");
};
```

alterazioni del linguaggio

```
#define if int
#define float while
#define switch for
#define return main
#define void(x) printf(x)

return() {
    if a=1, b, c=3;
    float(a<c) {
        switch( b=0; b<2 ;b++ ) {
            void("che fa?\n");
        }
        a++;
    }
}
```



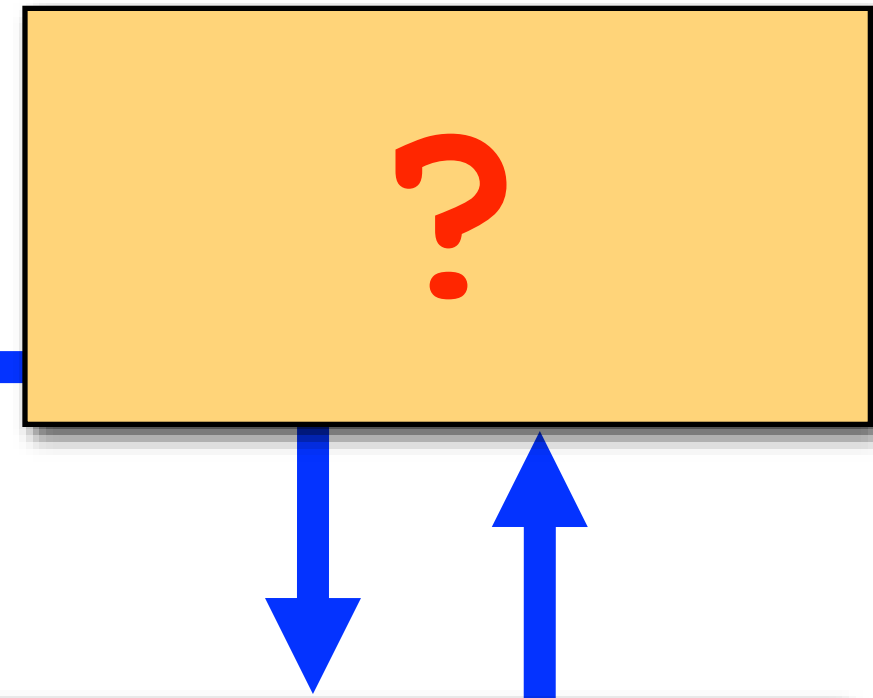
```
main() {
}
}
```

alterazioni del linguaggio

```
#define if int
#define void while
#define switch for
#define return main
#define float(x) printf(x)

return() {
    if a=1, b, c=3;
    void(a<c) {
        switch( b=0; b<2 ;b++ ) {
            float("che fa?\n");
        }
        a++;
    }
}
```

```
main() {
    int a=1, b, c=3;
    while(a<c) {
        for( b=0; b<2 ;b++ ) {
            printf("che fa?\n");
        }
        a++;
    }
}
```



alterazioni del linguaggio

```
#define if int
#define void while
#define switch for
#define return main
#define float(x) printf(x)

return() {
    if a=1, b, c=3;
    void(a<c) {
        switch( b=0; b<2 ;b++ ) {
            float("che fa?\n");
        }
        a++;
    }
}
```

che fa?
che fa?
che fa?
che fa?

```
main() {
    int a=1, b, c=3;
    while(a<c) {
        for( b=0; b<2 ;b++ ) {
            printf("che fa?\n");
        }
        a++;
    }
}
```

```
#define X
#define XX
#define XXX
#define XXXX
#define XXXXX
#define XXXXXX
#define orfa for
#define XXXXXXXXX
#define archa char
#define ainma main
#define etcharga getchar
#define utcharpa putchar

        X
      X X
    X   X
  X     X
X       X
X       X
X       X
X   X   X
X   XX   X
X   XXX   X   XXXXXXXXXX   X   XXX   X
X   XXX   X   XXXX   XXXX   X   XXX   X
X   XXXX   X   XX   ainma(){ archa   XX   XXXX   X
X   XXXX   X   oink[9],*igpa,   X   XXXX   X
X   XXXXXX   atinla=etcharga(),iocccwa XXXXXX   X
X   XXXX   ,apca='A',owla='a',umna=26   XXXX   X
X   XXX   ; orfa(; (atinla+1)&&!(((   XXX   X
X   XX   atinla-apca)*(apca+umna-atinla)   XX   X
X   X   >=0)+((atinla-owla)*(owla+umna-   X   X
X   atinla)>=0))); utcharpa(atinla),   X
X   X   atinla=etcharga()); orfa(; atinla+1;   X   X
X   X   ){ orfa(   igpa=oink   ,iocccwa=(   X   X
X   X   (atinla-   XXX   apca)*(   XXX   apca+umna-   X   X
X   X   atinla)>=0)   XXX   XXX   ; (((   X
X   atinla-apca   XXXXX   XXXXXXXX   XXXXX   )*(apca+   X
X   umna-atinla   XXXXXXXX   )>=0)   XXXXXXXX   +((atinla-   X
X   owla)*(owla+   XXXX   umna-   XXXX   atinla)>=0))   X
X   &&"-Pig-"   XX   "Lat-in"   XX   "COB-fus"   X
X   "ca-tion!!!"[   X   (((atinla-   X   apca)*(apca+   X
X   umna-atinla)   X   >=0)?atinla-   X   apca+owla:   X
X   atinla-owla   X   ]-'-')||((igpa==   X   oink)&&!(*(   X
X   igpa++)='w')   X   )|||   X   (*(   X   igpa   ++)=owla); *   X
X   (igpa++)=((   X   (   XXX   XXX   X   atinla-apca   X
X   )*(apca+   X   umna   XXX - XXX   X   atinla)>=0)   X
X   ?atinla-   X   apca   XXX + XXX   owla   X   :atinla),   X
X   atinla=   X   X   X   X   etcharga())   X
X   ; orfa(   X   atinla=iocccwa?((   X   (atinla-   X
X   owla)*(owla+   X   umna-atinla)>=0   X   )?atinla-   X
X   owla+apca:   X   atinla):   X   atinla; (((   X
X   atinla-apca)*   X   (apca+umna-   X   atinla)>=0)+(   X
X   (atinla-owla)*   X   (owla+   X   umna-atinla)>=   X
X   0)); utcharpa(   XX   XX   atinla),atinla   X
X   =etcharga());   XXXXXXXX   orfa(*igpa=0,   X
X   igpa=oink; *   igpa; utcharpa(   X
X   X   *(igpa++)); orfa(; (atinla+1)&&!(((   X
X   X   atinla-apca   )*(apca+   X
X   X   umna-   XXXXX   XXXXX   atinla)>=0   X
X   X   )+((   XXXXX   atinla-   X
X   XX   owla)*(   owla+umna-   XX
X   XX   atinla)>=0))))); utcharpa   XX
X   XX   (atinla),atinla=   XX
X   XX   etcharga()); }   XX
X   XXXX   }   XXXX
X   XXXXXXXXXX
```



#include e #define

#include

- conosciamo direttive come le seguenti:

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

- un'altra forma di inclusione si ottiene con:

```
#include "filename"
```

- incontrando tale riga, il preprocessore la sostituisce con una copia del contenuto del file indicato.

la ricerca del file avviene solo negli altri punti, e non nella directory corrente

la ricerca del file avviene prima nella directory corrente e poi in altri punti, dipendenti dal sistema, per esempio in Unix in `/usr/include/`

#define

```
#define identifier token_stringopt
```

```
#define SECS_PER_DAY (60*60*24)
```

- il preprocessore sostituisce ogni occorrenza di ***identifier*** con ***token_string*_{opt}**, **eccetto** le occorrenze all'interno di stringhe tra **virgolette**
 - il preprocessore sostituisce con (60*60*24) ogni occorrenza della costante SECS_PER_DAY per il resto del file
- l'utilizzo di ***#define*** può migliorare la **chiarezza** e la **portabilità** dei programmi: le costanti simboliche migliorano la documentazione con **nomi mnemonici** e **significativi**
minor **numero di interventi per manutenzione** e modifiche del codice

macro con parametri

```
#define identifier(identifier, ..., identifier)
                                token_stringopt
```

- non ci possono essere spazi fra il primo identificatore e l'apertura della tonda
- nell'elenco dei parametri possono esserci zero, uno o più identificatori

```
#define SQ(x) ((x) * (x))
```

l'identificatore **x** è un parametro che viene sostituito nel testo successivo

NB: la sostituzione è un rimpiazzamento di stringhe, e avviene **senza valutare la correttezza sintattica**

esempi

```
#define SQ(x) ((x) * (x))  
int i = 3;
```

```
printf("SQ(%d) = %d", i, SQ(i));
```

[rimpiazzamento]

((3) * (3))

```
$ SQ(3) = 9
```

esempi

```
#define SQ(x) ((x) * (x))  
int i = 3;
```

```
printf("SQ(%d) = %d", i, SQ(i));
```

((3) * (3))

```
$ SQ(3) = 9
```

```
printf("SQ(7+%d) = %d", i, SQ(7+i));
```

((7+i) * (7+i))

```
$ SQ(7+3) = 100
```


esempi (di errori)

```
#define SQ(x) x * x  
int i = 3;
```

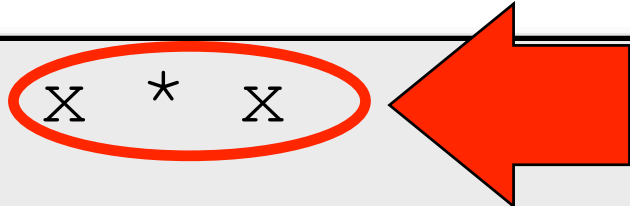
```
printf("SQ(%d) = %d", i, SQ(i));
```

3 * 3

```
$ SQ(3) = 9
```

esempi (di errori)

```
#define SQ(x) x * x  
int i = 3;
```



```
printf("SQ(%d) = %d", i, SQ(i));
```

3 * 3

```
$ SQ(3) = 9
```

```
printf("SQ(7+%d) = %d", i, SQ(7+i));
```

... $7+i * 7+i$ è ben diverso da $(7+i) * (7+i)$?????

```
$ SQ(7+3) = 31
```

esempi

- in generale le macro possono essere utilizzate per sostituire le chiamate di funzione con codice inline, che risulta più efficiente

```
#define min(x, y) ((x) < (y)) ? (x) : (y)
```

- parametri di min possono essere espressioni arbitrarie, purché di tipi compatibili
- per esempio, per determinare il minimo fra 4 valori:

```
#define min4(w, x, y, z) min(min(w, x), min(y, z))
```

- parametri di min possono essere espressioni arbitrarie (incluse funzioni!), purché di tipi compatibili

macro predefined

macro predefinedite

macro

valore

__LINE__ A decimal constant representing the current line number.

__FILE__ A string representing the current name of the source code file.

__DATE__ A string representing the current date when compiling began for the current source file. It is in the format "mmm dd yyyy", the same as what is generated by the asctime function.

__TIME__ A string literal representing the current time when cimpiling began for the current source file. It is in the format "hh:mm:ss", the same as what is generated by the asctime function.

__STDC__ The decimal constant 1. Used to indicate if this is a standard C compiler.

esempio

```
char *time,  
      *date,  
      *file;  
int line, stdc;
```

```
time = __TIME__;  
date = __DATE__;  
file = __FILE__;  
line = __LINE__;  
stdc = __STDC__;
```

```
printf("__TIME__ : %s\n", time);  
printf("__DATE__ = %s\n", date);  
printf("__FILE__ = %s\n", file);  
printf("__LINE__ = %d\n", line);  
printf("__STDC__ = %d\n", stdc);
```

```
$ ./prova  
__TIME__ = 02:55:57  
__DATE__ = Nov 1 2022  
__FILE__ = prova.c  
__LINE__ = 53  
__STDC__ = 1  
$
```

output del preprocessore

- invece di semplificare le cose, un uso eccessivo delle macro **complica il *debug*** dei programmi
- istruzione **gcc -E nome_file.c**
- dalla **documentazione di gcc:**

-E option

Stop after the preprocessing stage; do not run the compiler proper. The **output** is in the form of **preprocessed source code**, which is sent to the standard output or to a file named with the **-o** option

compilazione condizionale

compilazione condizionale

<code>#if</code>	<code>constant_integral_expression</code>
<code>#ifdef</code>	<code>identifier</code>
<code>#ifndef</code>	<code>identifier</code>
<code>#elif</code>	<code>constant_integral_expression</code>
<code>#else</code>	<code>-</code>
<code>#endif</code>	<code>-</code>
<code>#undef</code>	<code>identifier</code>

- l'espressione costante intera utilizzata nelle direttive al preprocessore non può contenere l'operatore ***sizeof*** o un cast

esempio

- delle strutture di controllo a disposizione del preprocessore...

```
#define DEBUG 0

#ifdef DEBUG
    printf("DEBUG definito\n");
    #if DEBUG
        printf("DEBUG diverso da zero\n");
    #else
        printf("DEBUG zero\n");
    #endif
#else
    printf("DEBUG non definito\n");
#endif
```

esempio

```
#define DEBUG 1

...
list add_node(list list_ptr, int value) {
#ifdef DEBUG
    assert(list_ptr != NULL);
#endif
    node* new_elem = malloc(sizeof(node));

    ...

    return list_ptr;
}
```

programmazione modulare

suddivisione del codice in moduli

```
int somma(int primo, int secondo);  
int moltiplica(int primo, int secondo);
```

```
int main() {  
    int i = 3;  
    int j = 5;
```

possiamo suddividere i programmi con una struttura complessa in **moduli riusabili**

```
    printf("i+j = %d\n", somma(i,j) );  
    printf("i*j = %d\n", moltiplica(i,j) );  
} // : : : : : : : : : : : : : : : : : :
```

```
int moltiplica(int primo, int secondo) {  
    return (primo*secondo);  
} // : : : : : : : : : : : : : : : : : :
```

```
int somma(int primo, int secondo) {  
    return (primo+secondo);  
}
```

file_completo.c

suddivisione del codice in moduli

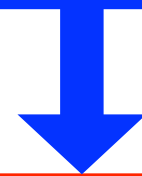
```
int main(void) {  
    int i = 3;  
    int j = 5;  
  
    printf("i+j = %d\n", somma(i,j) );  
    printf("i*j = %d\n", moltiplica(i,j) );  
}
```

modulo_di_test.c

```
int moltiplica(int primo, int secondo) {  
    return (primo*secondo);  
}  
  
int somma(int primo, int secondo) {  
    return (primo+secondo);  
}
```

mie_funzioni.c

problema: il file
[modulo_di_test.c](#)
deve conoscere i
prototipi di **somma** e
moltiplica



potremmo aggiungere
a mano i prototipi nel
main(), ma ad ogni
modifica delle funzioni
[saremmo costretti a](#)
[modificare a mano i](#)
[file che le usano...](#)

header file

```
#include "mio_header_file.h"

int main() {
    int i = 3;
    int j = 5;

    printf("i+j = %d\n", somma(i,j) );
    printf("i*j = %d\n", moltiplica(i,j) );
}
```

modulo_di_test.c

```
// qst file è mio_header_file.h
// i miei prototipi sono memorizzati qui!
//
int moltiplica(int primo, int secondo);
int somma(int primo, int secondo);
```

mio_header_file.h

problema:
modulo_di_test.c deve conoscere i prototipi di **somma** e **moltiplica**

oppure potremmo

- memorizzare tutti i prototipi in un singolo file header (intestazione) con estensione .h
- includere tale file all'inizio dei moduli che necessitano dei prototipi definiti nell'header file

header file

```
#include "mio_header_file.h"
```

```
int main() {  
    int i = 3;  
    int j = 5;  
    printf("i+j = %d\n", somma(i,j) );  
    printf("i*j = %d\n", moltiplica(i,j) );  
}
```

con la direttiva *#include*
richiediamo al preprocessore di
rimpiazzare il token

```
#include "mio_header_file.h"
```

con il contenuto del file

```
mio_header_file.h
```

modulo_di_test.c

```
// qst file è mio_header_file.h  
// i miei prototipi sono memorizzati qui  
//
```

```
int moltiplica(int primo, int secondo);
```

```
int somma(int primo, int secondo);
```

```
// inoltre possiamo aggiungere anche
```

```
// l'implementazione delle funzioni
```

```
// online...
```

```
int moltiplica(int primo, int secondo) {
```

```
    return (primo+secondo);}
```

```
int somma(int primo, int secondo) {
```

```
    return (primo*secondo);}
```

mio_header_file.h

gli header files possono contenere:

- **prototipi** di funzioni
- **definizioni** di tipi (strutture, unioni, tipi enumerativi, typedef)
- **macro** *#define*
- **istruzioni** *#pragma* per il compilatore
- variabili **globali**
 - crea una variabile globale in ogni modulo che include il file header a meno che la variabile sia dichiarata **extern**
- implementazione di **funzioni inline**

le chiamate di funzioni inline sono direttamente rimpiazzati dal corpo delle funzioni stesse

più moduli...

```
#include "somma.h"
int somma(int primo, int secondo) {
    return (primo+secondo);
}
```

somma.c

```
#include "prodotto.h"
int moltiplica(int primo, int secondo) {
    return (primo*secondo);
}
```

prodotto.c

```
int somma(int primo, int secondo);
```

somma.h

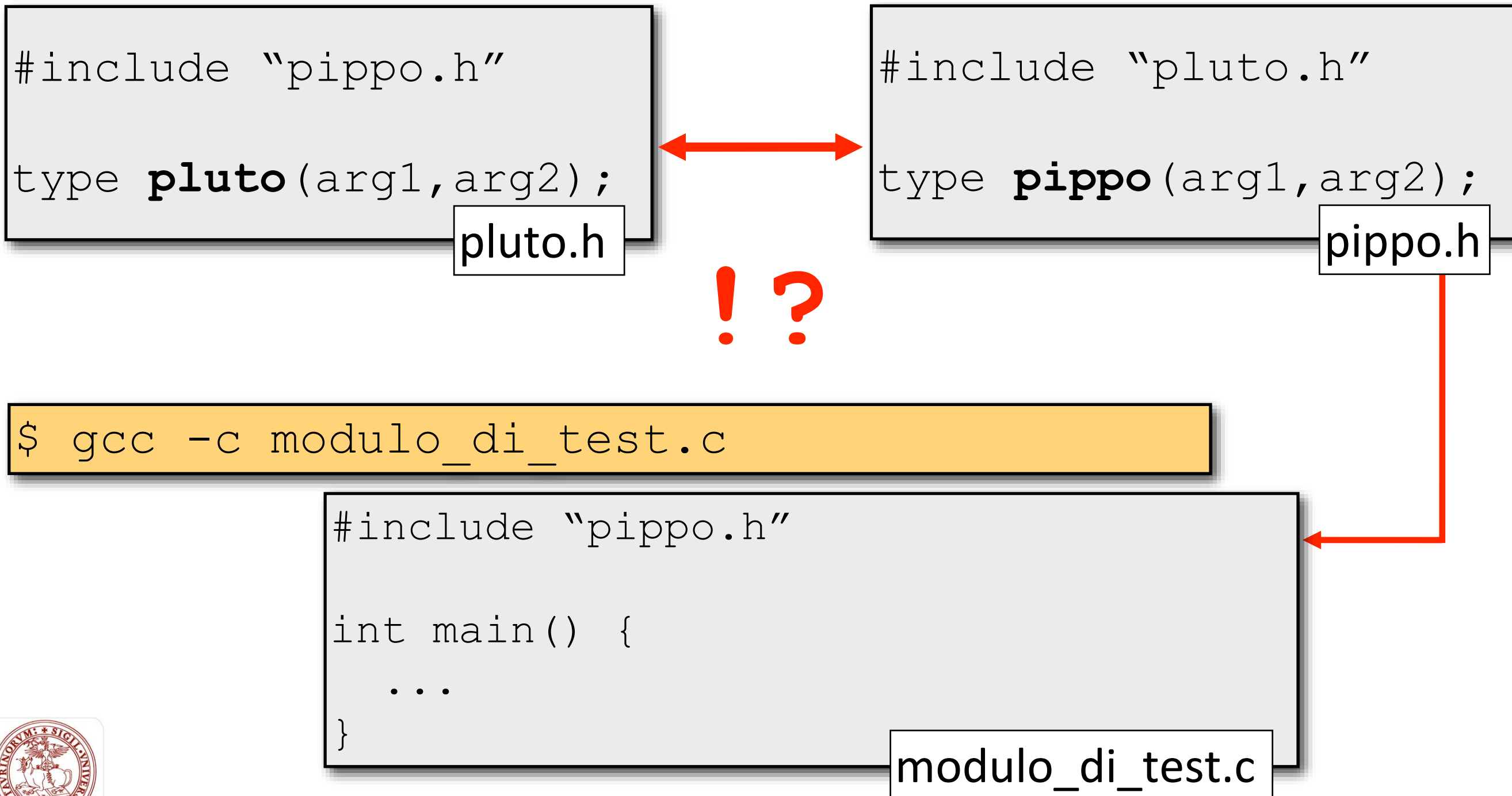
```
int moltiplica(int primo, int secondo);
```

prodotto.h

```
#include "somma.h"
#include "prodotto.h"
```

```
int main() {
    int i = 3;
    int j = 5;
    printf("i+j = %d\n", somma(i,j));
    printf("i*j = %d\n", moltiplica(i,j));
}
```

ciclicità nelle dipendenze



ciclicità nelle dipendenze

```
#include "pippo.h"

type pluto(arg1, arg2);
```

pluto.h

! ?



```
#include "pluto.h"

type pippo(arg1, arg2);
```

pippo.h

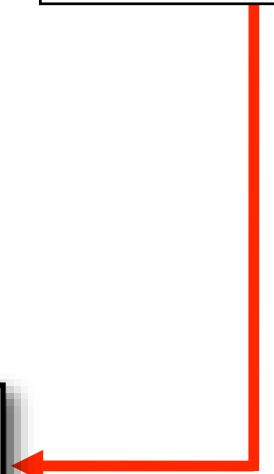
loop *infinito* in fase di compilazione!

```
$ gcc -c modulo_di_test.c
```

```
#include "pippo.h"

int main() {
    ...
}
```

modulo_di_test.c



per evitare inclusioni multiple

```
#ifndef __PLUTO_H__  
#define __PLUTO_H__  
  
#include "pippo.h"  
  
type pluto(arg1, arg2);  
#endif
```

pluto.h

```
#ifndef __PIPPO_H__  
#define __PIPPO_H__  
  
#include "pluto.h"  
  
type pippo(arg1, arg2);  
#endif
```

pippo.h

possiamo utilizzare *#ifndef* e *#define* per fare sì che il contenuto sia incluso una sola volta, quando è raggiunta la direttiva *#include*.

solo se `__PLUTO_H__` e `__PIPPO_H__` non sono ancora definiti sono inserite le righe fra *#ifndef* e *#endif*

questo meccanismo impedisce ulteriori inclusioni

compilazione dei moduli

```
$ gcc -c somma.c  
$ gcc -c prodotto.c  
$ gcc -c modulo_di_test.c  
$ gcc -o modulo_di_test *.c
```

} creazione di codice
oggetto

creazione dell'eseguibile

- i singoli moduli, cioè i file **.c** possono essere compilati in **file oggetto** specificando il parametro **-c**
- esiste una **dipendenza** fra i file **.h** e il file **.c** che li include: **se un file di intestazione cambia, allora i moduli dipendenti devono essere ricompilati**

compilazione separata

- se il progetto consiste di centinaia di files, può tuttavia essere utile la compilazione separata: ma come determinare **quali moduli ricompilare**?
- dimenticando di ricompilarne qualcuno, può capitare che il linker generi errori nel tentare di utilizzare un file oggetto obsoleto;
- peggio, **se la *signature* non cambia**, l'errore non sarà segnalato preventivamente, provocando **errori a runtime**
- possiamo risolvere tali problemi con l'**utility make**.

l'utility *make*

- la make-utility è uno strumento che può essere utilizzato per **automatizzare** il processo di compilazione
- in generale è **più flessibile** degli ambienti integrati (IDE, integrated development environments) come MS Visual .NET, Xcode, NetBeans, o Borland C++
- si basa sull'utilizzo di un file (***makefile***) che **descrive le dipendenze** presenti nel progetto
- è quindi possibile utilizzare il **comando make** che si avvale della ***marcatatura temporale*** dei files e **ricompila i target file che sono più vecchi dei sorgenti**

make

- il **Makefile** elenca un insieme di target, le regole per la compilazione e l'istruzione da eseguire per compilare a partire dai sorgenti

riga di dipendenza, che indica da quali file oggetto dipende il file **target**; inizia dalla prima colonna della riga

```
nome_target: file1.o file2.o file_n.o
    gcc -o nome_target file1.o file2.o file_n.o
file1.o: file1.c file1.h
    gcc -c file1.c
...
file_n.o: file_n.c file_n.h
    gcc -c file_n.c
```

riga d'azione o di comando, che indica come il programma deve essere compilato nel caso sia stato modificato almeno uno dei file .o

- deve iniziare con una tabulazione
- dopo una riga di dipendenza è possibile specificare più di una riga d'azione

make

```
nome_target: file1.o file2.o file_n.o
    gcc -o nome_target file1.o file2.o file_n.o
...
clean:
    rm -f *.o
```

ulteriore ***target***, che è possibile invocare per rimuovere tutti i file oggetto

- nell'invocare make da linea di comando è possibile specificare quale target compilare

```
$ make file1.o    //crea solo file1.o
```

- se viene invocato senza opzione dalla linea di comando, utilizza il **target di default**, il primo specificato nel makefile

```
$ make //crea file1.o file2.o file_n.o e nome_target
```

- per ricompilare cancellando tutti i file oggetto

```
$ make clean
```



gli argomenti del *main*

Command-Line arguments (*argc*, *argv*)

- the command-line arguments (the separate words parsed by the shell) are made available via two arguments to the function *main()*
 - the first argument, *int argc*, indicates **how many command-line arguments there are**.
 - the second argument, *char *argv[]*, is an **array of pointers** to the command-line arguments, each of which is a null-terminated character string. the first of these strings, in *argv[0]*, is the name of the program itself.

```

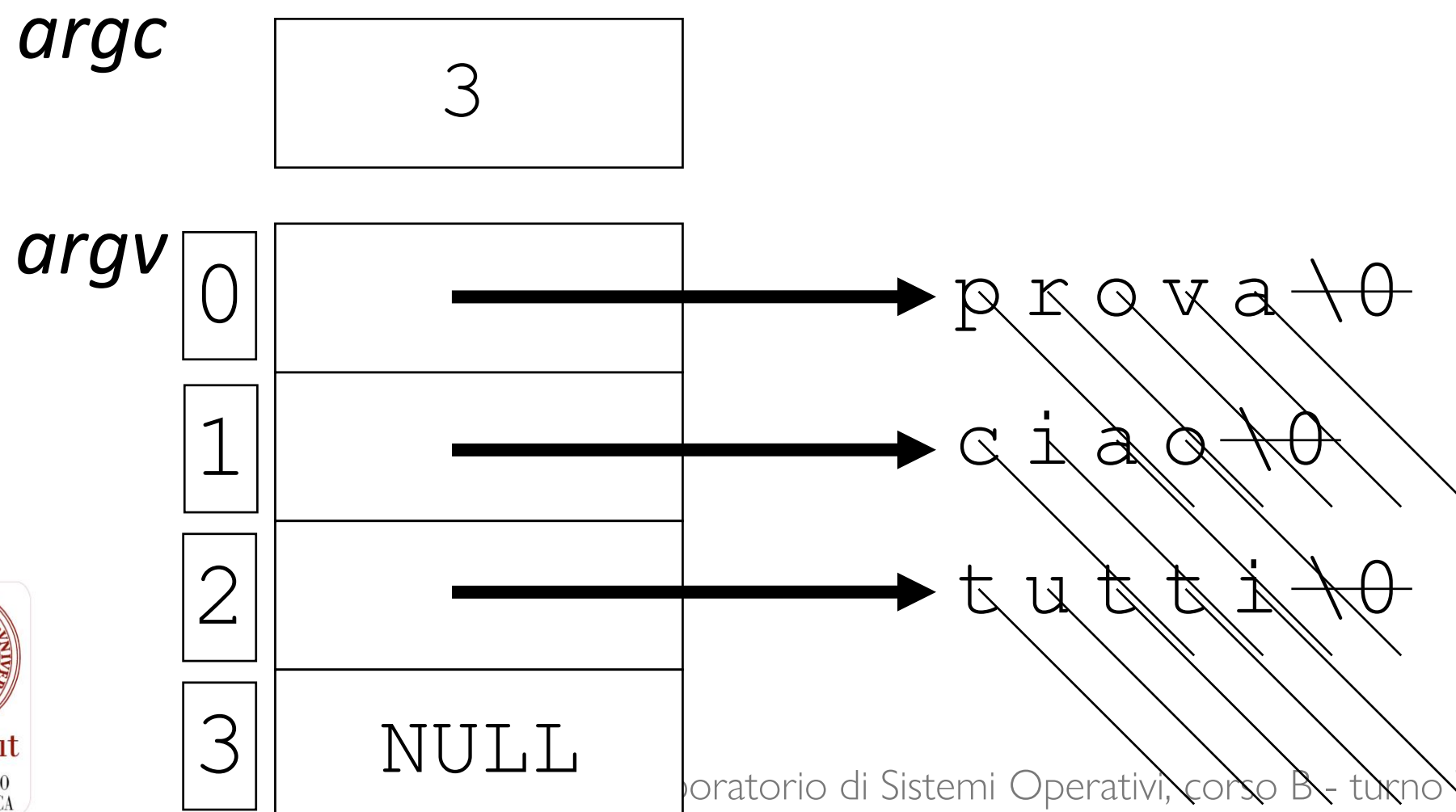
int main(int argc, char *argv[]) {
    int j;

    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);

    exit(EXIT_SUCCESS);
}

```

\$./prova ciao tutti



Environment List

- Each process has an **associated array of strings** called the ***environment list***, or simply the ***environment***. Each of these strings is a definition of the form ***name=value***.
 - Thus, the ***environment*** represents a set of ***name-value pairs*** that can be used to hold arbitrary information.
 - The names in the list are referred to as **environment variables**.

Environment List

- When a **new process** is created, it **inherits** a **copy of its parent's environment**.
 - This is a primitive and frequently used form of interprocess communication.
- A common use of environment variables is in the shell.
 - By placing values in its own environment, the shell can ensure that these values are passed to the processes that it creates to execute user commands.

Environment List

- For example, the environment variable SHELL is set to be the *pathname* of the shell program itself.
- Many programs interpret this variable as the name of the shell that should be executed if the program needs to execute a shell.

```
$ echo $SHELL  
/bin/bash
```


Environment List

```
$ mia_var=valore_var // creazione variabile
$ export mia_var      // aggiunta della
                       // variabile nell'
                       // environment del
                       // processo
```

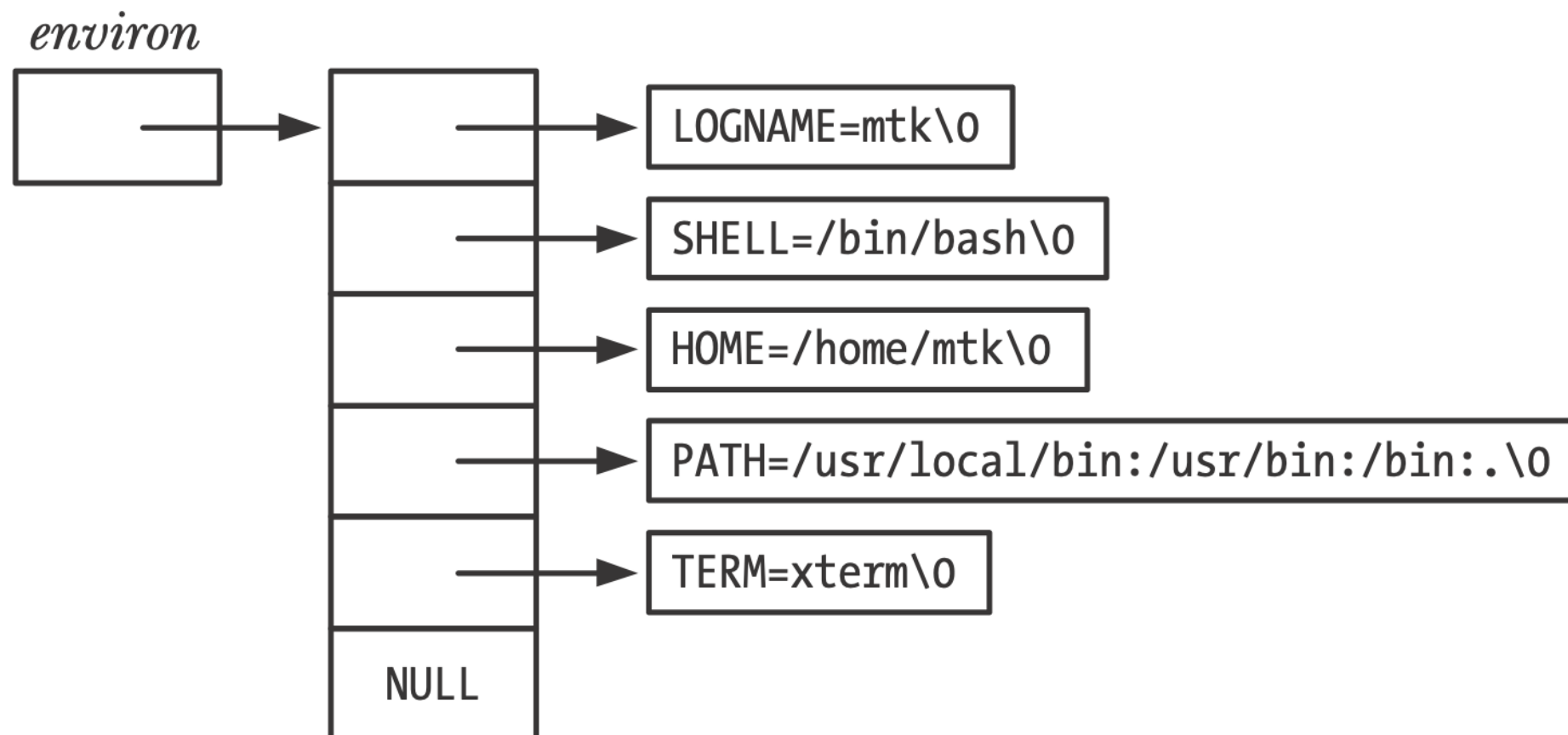
```
// processo
```

```
// environment del processo
```

- In most shells, a value can be added to the environment using the ***export*** command
- The ***printenv*** command displays the **current environment list**.

Accessing environ from C programs

- Within a C program, the environment list can be accessed using the global variable ***char **environ***. Like ***argv***, ***environ*** points to a NULL-terminated list of pointers to null-terminated strings.



Accessing environ from C programs

- Within a C program, the environment list can be accessed using the global variable char *****environ***.

```
#include <unistd.h>

extern char **environ;
// di qui in poi è possibile utilizzare
// environ
```

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

extern char ** environ;
// di qui in poi è possibile utilizzare
// environ
int main(int argc, char** argv) {

    char** cursor;
    for(cursor=environ; *cursor!= NULL; ++cursor) {
        puts(*cursor);
    }

    return (EXIT_SUCCESS);
}
```