TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Linguaggio C: Puntatori

Valeria Cardellini

Corso di Calcolatori Elettronici
A.A. 2019/20

Argomenti

- Definizione ed uso di puntatori
- Passaggio di parametri tramite puntatori
- Relazione tra puntatori, vettori e matrici
- Tipo void* e casting sui puntatori
- Gestione dinamica della memoria

Puntatori

- In C si possono gestire gli indirizzi di memoria attraverso le variabili di tipo puntatore
- Esempio

```
int a = 5;
```

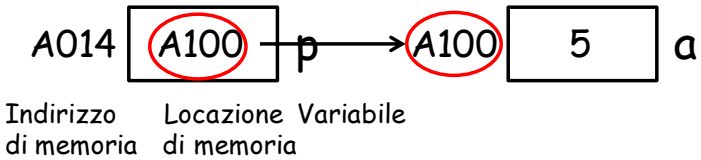
  - Proprietà della variabile a
    - ✓ Nome: a
    - ✓ Tipo: int
    - ✓ Valore: 5
    - ✓ **Indirizzo**: 0xA100
  - &a: **&** è l'**operatore indirizzo-di** applicato alla variabile a
  - Gli indirizzi si utilizzano nelle variabili puntatore

Puntatori

- I valori delle variabili di tipo puntatore sono indirizzi di memoria
  - Valori numerici che fanno riferimento a specifiche locazioni di memoria
- Esempio

```
int a = 5;
int *p;
p = &a;
```

  - p è un puntatore ad a ovvero p punta alla variabile a



Puntatori: dichiarazione

- Esempio dichiarazione puntatore:

```
int *p;
```

- Proprietà della variabile p
  - Nome: p
  - Tipo: puntatore ad int (indirizzo di intero)
  - Valore: se non inizializzata, è casuale (oppure NULL)
  - Indirizzo: fissato
- Sintassi della dichiarazione di variabili puntatore

  tipo *nome_var1, …, *nome_varn;
- Esempio:

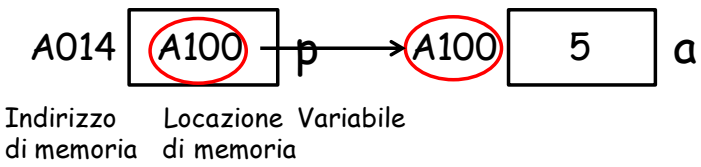
```
int *p1, i, *p2, j; /* p1 e p2: variabili puntatore a
                       intero */
float *pf, g;       /* pf: variabile puntatore a float */
```

  - Attenzione: nelle dichiarazioni multiple (es. `float *pf, g;`) g non è un puntatore!

Operatori indirizzo-di e dereferenziazione

- Una variabile puntatore può essere inizializzata utilizzando l'**operatore indirizzo-di &** (ampersand)
- Esempio:

```
int a, *p;
p = &a;
```



- **Operatore di dereferenziazione *** (dereferencing o indirizzamento indiretto)
  - Applicato ad una variabile puntatore restituisce il valore della locazione di memoria puntata
  - Esempio:

```
int *p, a = 10, b;
p = &a;   /* in p indirizzo di a */
b = *p;   /* in b valore contenuto nella locazione di
             memoria puntata da p */
```

  - Nota: se p è di tipo `int *` allora `*p` è di tipo `int`

Operatori indirizzo-di e dereferenziazione

- L'operatore di dereferenziazione *
  - Usato come rvalue (right value, la parte alla destra dell'operatore =) esegue un'operazione di estrazione
  - Esempio:

```
int a, *p;
…
a = *p;  /* assegna ad a il valore della variabile
            puntata da p */
```

  - Usato come lvalue (left value,la parte alla sinistra dell'operatore =) esegue un'operazione di inserimento
  - Esempio:

```
int a, *p;
…
*p = a;  /* assegna il valore di a alla variabile
            puntata da p */
```


Operatori indirizzo-di e dereferenziazione

- Non confondere:
  - l'operatore (unario) di indirizzamento indiretto * con l'operatore di moltiplicazione (binario)
  - * in una dichiarazione: serve per dichiarare una variabile di tipo puntatore (es. `int *p;`)
  - * in una espressione: è l'operatore di dereferenziazione (es. `a = *p;`)
- Proprietà degli operatori * e &
  - Priorità più elevata degli operatori binari
  - * associativo a destra: `**p` equivale a `*(*p)`
  - & applicabile solo a variabili, ad es. &a
  - * e & sono uno inverso dell'altro
    - ✓ Data la variabile `int a;` allora *&a equivale ad a
    - ✓ Data la variabile `int *p;` allora &*p equivale a p

Operatori indirizzo-di e dereferenziazione

- I puntatori si possono stampare (in esadecimale) con lo specificatore di formato %p
- Esempio

Codice sorgente: pointer.c

```
int a = 5;
int *p;

p = &a;

printf("indirizzo di a: %p\n", &a);
printf("valore di p: %p\n", p);
printf("valore di &*p: %p\n", &*p);
printf("valore di a: %d\n", a);
printf("valore di *p: %d\n", *p);
printf("valore di *&a: %d\n", *&a);
```


Inizializzazione di puntatori

- Come tutte le variabili, i puntatori devono essere inizializzati prima di essere usati
- Errore: dereferenziare una variabile puntatore non inizializzata
  - Esempio

```
int a = 5;
int *p;
printf("val. di *p: %d", *p);
```

```
warning: variable 'p' is uninitialized when used here
      [-Wuninitialized]
   printf("valore di *p: %d\n", *p);
                                ^
note: initialize the variable 'p' to silence this warning
   int *p;
           ^
         = NULL
```

- Attenzione: la dichiarazione di una variabile puntatore non alloca memoria per la variabile puntata
  - Prima di accedere alla variabile puntata bisogna allocare esplicitamente memoria (vedremo come)

Tipo di variabile puntatore

- Il tipo di una variabile puntatore è "puntatore a tipo"
  - Il suo valore è un indirizzo
  - Attenzione: il tipo puntatore è un indirizzo, non un intero!
- Due variabili tipo puntatore che si riferiscono a tipi diversi non sono compatibili tra loro
- Esempio:

```
int *pi;
float *pf;
pf = pi;
```

```
warning: incompatible pointer types assigning to 'float *'
      from 'int *' [-Wincompatible-pointer-types]
   pf = pi;
```

- Perché il C distingue tra puntatori di tipo diverso?
  - Per determinare a tempo di compilazione il tipo della variabile puntatore
  - Vedremo che esiste il puntatore a **`void`**

Operatore `sizeof()`

- L'operatore `sizeof()` fornisce l'occupazione di memoria in byte di una variabile
  - Può essere applicato anche ad un tipo
  - Es.: `sizeof(int)`
- Tutti i puntatori sono indirizzi!
  - Spazio di memorizzazione occupato da un puntatore = spazio di memorizzazione di un indirizzo
- L'oggetto puntato ha dimensione del tipo puntato

Esempio: operatore sizeof()

Codice sorgente: pointer_sizeof.c

```
#include <stdio.h>

int main(void)
{
 char *pc;
 int *pi;
 double *pd;

 printf("%lu %lu %lu\n", sizeof(pc), sizeof(pi), sizeof(pd));
 printf("%lu %lu %lu\n", sizeof(char *), sizeof(int *), sizeof(double *));

 printf("%lu %lu %lu\n", sizeof(*pc), sizeof(*pi), sizeof(*pd));
 printf("%lu %lu %lu\n", sizeof(char), sizeof(int), sizeof(double));

 return 0;
}
```


Operazioni sui puntatori

- Sui puntatori si possono effettuare diverse operazioni
- Dereferenziazione:

```
int *p, i;
i = *p;
```

- Assegnamento:

```
int *p, *q;
p = q;
```

- Confronto:

```
int *p, *q;
if (p == q) … /* Attenzione: diverso da if (*p == *q) */
if (p > q) …
```

Operazioni sui puntatori

- Operazioni aritmetiche:
  - Incremento (++) o decremento (--)
  - Esempio:

```
int *p;
...
p++;
```

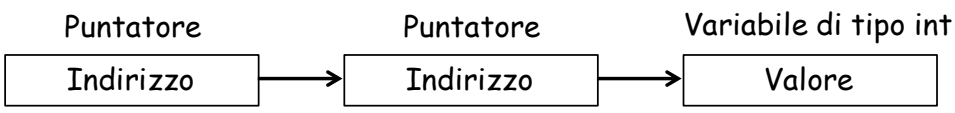
    - ✓ L'operazione di incremento `p++` consente di puntare alla successiva locazione di memoria di tipo `int`
    - ✓ Stampando il valore di `p` dopo l'incremento, esso risulterà incrementato di `sizeof(int)`, ad es.

```
p prima incremento: 0x7ffee7b5f900
p dopo incremento: 0x7ffee7b5f904
```

  - Somma (+=) o sottrazione (-=) di un intero
  - Sottrazione di un puntatore da un altro

Puntatori a puntatori

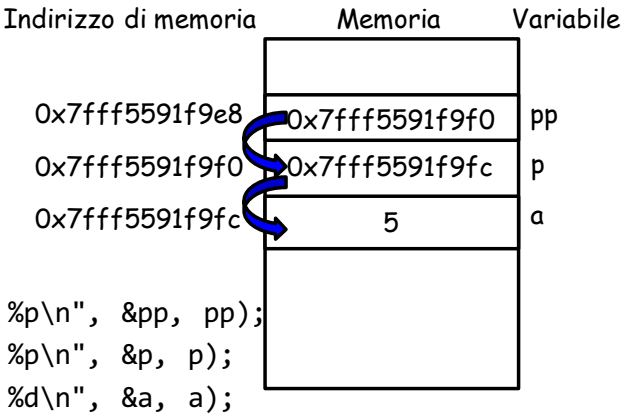
- Un puntatore è una variabile che contiene un indirizzo
- Poiché il puntatore è una variabile con un tipo, il suo indirizzo è (seguendo la regola generale) un puntatore a un puntatore
  - Indirizzamento indiretto
- Esempio:
  - L'indirizzo di un puntatore di tipo `int` è un puntatore a puntatore a intero
  - Si scrive `int **`



Esempio: puntatori a puntatori

```
int main(void) {
  int a, *p, **pp;

  a = 5;
  p = &a;        /* Indirizzo di a */
  pp = &p;       /* Indirizzo di p */
  printf("pp:   indirizzo = %p, valore = %p\n", &pp, pp);
  printf("p:    indirizzo = %p, valore = %p\n", &p, p);
  printf("a:    indirizzo = %p, valore = %d\n", &a, a);
  printf("*p:   valore = %d\n", *p);
  printf("**pp: valore = %d\n", **pp);
  printf("Modifico il valore di **pp\n");
  **pp = 10;
  printf("a:   indirizzo = %p, valore = %d\n", &a, a);
  return 0;
}
```



Codice sorgente: pointer2pointer.c

Passaggio di parametri

- Abbiamo già esaminato in C i limiti del passaggio dei parametri per valore, che ha una semantica per copia
  - Il parametro formale è una nuova variabile locale alla funzione
  - Al momento della chiamata il valore del parametro attuale viene copiato nel parametro formale
  - Le modifiche effettuate sul parametro formale non si riflettono sul parametro attuale
  - Esempio: `swap_val(x, y)` non produce effetti, cioè non restituisce gli argomenti con i valori scambiati

```
void swap_val(int a, int b) {
    int aux;
    aux = a;
    a = b; b = aux;
}
```

Simulare il passaggio di parametri per riferimento

- Come simulare in C il passaggio di parametri per riferimento?
- Idea: avendo una copia dell'indirizzo di una variabile, questa copia può essere usata per modificare la variabile
- Quindi, passando per valore alla funzione un tipo puntatore ad una variabile, la funzione può usare il puntatore per modificare la variabile
  - Se il parametro è un puntatore, la funzione chiamata non modifica il puntatore, ma può modificare la variabile da esso puntata

Esempio: passaggio di parametri di tipo puntatore

- Passaggio di parametri usando i puntatori

```
void swap(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Codice sorgente: pointer_swap.c

- A swap() sono passati gli indirizzi delle variabili da scambiare
  - Gli indirizzi non sono modificati
- swap modifica il valore puntato
  - *Come se* le variabili da scambiare fossero state passate per riferimento

Vantaggi del passaggio di parametri di tipo puntatore

- Il passaggio di parametri di tipo puntatore è particolarmente utile quando i dati che devono essere scambiati tra funzione chiamata e funzione chiamante sono voluminosi
- Passaggio di un puntatore:
  - Più efficiente in termini di occupazione di memoria e in termini di tempo
    - ✓ I dati non devono essere copiati nello stack
    - ✓ Infatti occorre la copia del solo puntatore ai dati, non di tutto l'insieme dei dati

Relazione tra puntatori e vettori

- In generale non sappiamo cosa contengono le locazioni di memoria adiacenti ad una data locazione
- L'unico caso in cui sappiamo quali sono e cosa contengono è quando utilizziamo delle variabili di tipo array
  - Consideriamo array ad 1 dimensione
- In C il nome di un array è l'indirizzo del primo elemento dell'array
- Esempio:

```
int vet[10]; /* vet e &vet[0] hanno lo stesso valore */
printf("%p %p", vet, &vet[0]);   /* stampa 2 volte lo
                                    stesso indirizzo */
```

Accesso agli elementi di un vettore

- Possiamo far puntare un puntatore al primo elemento di un vettore
- Esempio:

```
int vet[5], *pi;
pi = vet;
```

equivale a:

```
pi = &vet[0];
```

- Esempio:

```
int vet[5];
int *pi = vet;
*(pi + 3) = 28;
```

  - `pi+3` punta all'elemento di indice 3 del vettore
  - 3 viene detto offset (o scostamento) del puntatore

Accesso agli elementi di un vettore

- Esempio:

```
int vet[5];
int *pi = vet;
*(pi + 3) = 28;
```

  - `&vet[3]` equivale a `pi+3` che equivale a `vet+3`
  - `*&vet[3]` equivale a `*(pi+3)` che equivale a `*(vet+3)`
  - Inoltre `*&vet[3]` equivale a `vet[3]`
- Quindi in C `vet[3]` è un modo alternativo di scrivere `*(vet+3)`
- Notazioni per gli elementi di un vettore:
  - Notazione con puntatore e indice: `vet[3]`
  - Notazione con puntatore e offset: `*(vet+3)`

Accesso agli elementi di un vettore

- Riassumendo, si può accedere agli elementi di un vettore nel seguente modo:
- Esempio:

```
int vet[5] = {10, 15, 20, 25, 30};
int *pi = vet;
int offset = 3;
vet[offset] = 28;
*(vet + offset) = 28;
pi[offset] = 28;
*(pi + offset) = 28;
```


Accesso agli elementi di un vettore

- Esempio: è corretto questo frammento di codice?

```
int vet[10];
int *pi;
vet = pi;
vet++;
```

- No, perché vet è un puntatore costante, e quindi non può essere modificato

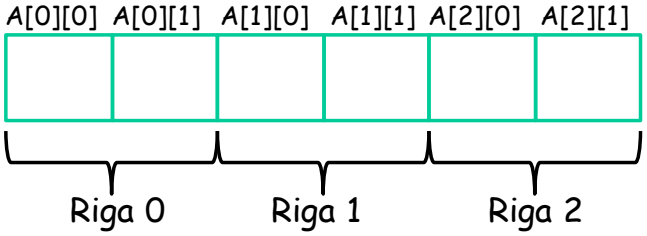
```
error: array type 'int [10]' is not assignable
vet = pi;
~~~ ^
error: cannot increment value of type 'int [10]'
vet++;
~~~^
2 errors generated.
```

Relazione tra matrici e puntatori

- In C le matrici sono allocate in memoria per righe

```
int A[3][2];   // Matrice con 3 righe e 2 colonne
```

- La matrice A è allocata per righe in 6 locazioni di memoria adiacenti:



- Nell'esempio `&(mat[i][j])` equivale a `mat[0] + 2*i + j`

```
A[1][1] = 10;      equivale a
*(A[0] + 2*1 + 1) = 10;
```

Numero di colonne

Passaggio di parametri di tipo vettore

- Quando si passa un vettore come parametro ad una funzione, in realtà si sta passando l'indirizzo dell'elemento di indice 0
- Il parametro formale deve essere di tipo puntatore al tipo degli elementi del vettore
- Di solito si passa la dimensione del vettore in un ulteriore parametro

Esempio: passaggio di parametri di tipo vettore

- Esempio:

```
void print_vec(int *v, int dim) {
  int i;
  for (i = 0; i < dim; i++)
    printf("v[%d]=%d ", i, v[i]);
  printf("\n”);
}

int main(void) {
  int vet[5] = {10, 15, 20, 25, 30};
  print_vec(vet, 5);
  return 0;
}
```

Codice sorgente: pointer_vec.c

Passaggio di parametri di tipo vettore

- Per evidenziare che il parametro formale è in realta un vettore (ovvero l'indirizzo dell'elemento di indice 0), di solito si usa la notazione nome-parametro[] invece di *nome-parametro
  - Esempio: print_vec(int v[], int dim) { ... }
- Si può anche specificare la dimensione del vettore tra [], ma viene ignorata
  - Esempio: print_vec (int v[5], int dim) { ... }
- Nel prototipo della funzione può anche mancare il nome del vettore.
  - Esempio: void print_vec(int [], int);
- Il passaggio di un vettore è in realtà un passaggio per indirizzo! La funzione può modificare gli elementi del vettore passato

Esempio: passaggio di parametri di tipo vettore

- Esempio: funzione che inverte un vettore

```
/* Inverte il vettore v di dimensione dim, versione
iterativa */
void reverse_vec(int v[], int dim)
{
  int temp; /* usata per scambiare due elementi del
               vettore */
  int i;

  for (i = 0; i < dim/2; i++) {  /* scambia v[i] con
                                    v[dim-1-i]*/
    temp = v[i];
    v[i] = v[dim-1-i];
    v[dim-1-i] = temp;
 } /* end for */
}
```

Codice sorgente: pointer_vec.c

Esempio: ordinamento per selezione

```
void selection_sort(int *v, int dim)
{
    int i=0, min, imin=0, j=0;

    for (i=0; i<dim-1; i++){
        min = v[i];
        imin = i;
        for (j=i+1; j<dim; j++) {
            if(v[j] < min) {
                min = v[j];
                imin = j;
            }
        } /* end inner for */
        v[imin] = v[i];
        v[i] = min;
    } /* end outer for */
}
```

Codice sorgente: order_vec.c

Esempio: ordinamento bubble sort

```
void bubble_sort(int *v, int dim)
{
  int i, j, comps;

  comps = dim-1;

  for (i = 0; i < comps; i++)
    for (j = 0; j < comps - i; j++) {
      if (v[j] > v[j+1])
        swap(&v[j], &v[j+1]);
    }
}
```

Codice sorgente: order_vec.c

Esempio: ordinamento bubble sort ottimizzato

```
void bubble_sort2(int *v, int dim)
{
  int i, comps, sorted = 0;

  comps = dim-1;
  while (!sorted) {
    sorted = 1;        /* set true for each pass */
    for  (i = 0; i < comps; i++) {
      if  (v[i] > v[i + 1]) {
        swap(&v[i], &v[i+1]);
        sorted = 0;   /* not yet sorted */
      }
    } /* end inner for */
    comps--; /* comps reduces on each pass */
  }    /* end while */
}
```

Codice sorgente: order_vec.c

Il puntatore NULL

- Le variabili di tipo puntatore possono assumere anche un valore speciale: `NULL`
  - Per specificare che la variabile non punta ad alcuna locazione di memoria
- NULL è una costante simbolica in genere definita in <stdio.h>
  - In C NULL in genere corrisponde a 0 ma si raccomanda di usare NULL, in particolare per verificare se ad una variabile puntatore non è associato uno specifico riferimento
- Non confondere variabili il cui valore è NULL con variabili non inizializzate
  - Una variabile non inizializzata non ha alcun valore, neanche NULL

Il puntatore NULL

- Il confronto con NULL può essere usato in una condizione di un'istruzione `if-else`, `for`, ...
- Esempio:

```
int *pt= NULL;
…
if (pt != NULL)
    *pt = 10;
```

Un puntatore generico: il tipo void*

- In generale, nel caso delle dichiarazioni dei tipi primitivi è indispensabile definire il tipo della variabile per consentire al compilatore di allocare la memoria necessaria
- Nel caso dei puntatori, la memoria per il puntatore è fissa e corrisponde alla dimensione di un indirizzo di memoria
  - Si può omettere la specifica del tipo della variabile puntata

```
void *pt;
int i;
pt = &i;
```


Il tipo void *

- Per un puntatore a void, ci sono delle restrizioni
  - E' un puntatore generico e **non** può essere dereferenziato
    - ✓ Se `void *pt;` non si può usare `*pt`
  - Non sono ammesse operazioni aritmetiche
  - Può essere necessario il casting (*tipo* *) per copiare un puntatore void in un puntatore non-void (dipende dal compilatore, in C++ è necessario)
- Non serve il casting (`void *`) per copiare un puntatore non-void in un puntatore void

```
void *pt;
int *pti;
pt = pti;
```

- NULL è definito come `(void *)0`

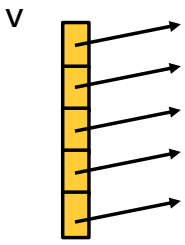
Casting su puntatori

- Anche nel caso delle variabili di tipo puntatore sono possibili conversioni esplicite (casting):

```
void *pt;
int i;
pt = &i;
int *pti;
pti = (int *) pt;      /* valore di pt convertito a
                          puntatore a int */
```


Array di puntatori

- Array di puntatori: gli elementi dell'array sono puntatori
  - Es.: `int *v[5];`
    - ✓ Definisce un vettore di 5 puntatori a `int`
    - ✓ `[]` ha priorità maggiore rispetto a `*`
    - ✓ I vari puntatori sono memorizzati in locazioni di memoria contigue, mentre le locazioni puntate non saranno generalmente contigue



Array di puntatori: inizializzazione

- Esempio di inizializzazione

```
int a, b, c;
int *v[5]={NULL};
v[0]=&a;
v[1]=&b;
v[2]=&c;
```

  - I valori di `v[3]` e `v[4]` sono `NULL` in quanto è stato inizializzato il primo elemento dell'array contestualmente alla dichiarazione

- Non confondere un array di puntatori con un puntatore ad array, ad es.

```
int (*q)[5];
```

q è un puntatore ad un array di 5 int

  - Notare l'uso di `()` per la precedenza

Gestione dinamica della memoria

- Finora abbiamo considerato il modello di allocazione della memoria tramite **stack**
  - Si basa sulle regole di campo d'azione (definito **staticamente**, cioè al momento della compilazione)
- Attraverso i puntatori, il C permette di utilizzare un altro modello di allocazione della memoria, che consente di definire la memoria **dinamicamente**, cioè al momento dell'esecuzione del programma
  - Particolarmente utile ed importante quando non sono note o prevedibili a priori le dimensioni dei dati

Allocazione dinamica della memoria

- Durante l'esecuzione, un programma può richiedere esplicitamente uno spazio di memoria per immagazzinare dati
- Può richiedere di rilasciare tale spazio quando non sarà più necessario
- In *C*:
  - funzione **`malloc`** per riservare (allocare) uno spazio di memoria
  - funzione **`free`** per rilasciare (deallocare) memoria
- Insieme all'operatore **`sizeof`** sono essenziali per l'allocazione dinamica della memoria

Funzione `malloc()`

- `malloc` (memory allocation) permette di allocare dinamicamente una porzione (chunk) di memoria

```
void* malloc(size_t size);
```

  - Alloca `size` byte di memoria
  - `size_t` è il tipo intero senza segno restituito dall'operatore `sizeof`
- Restituisce un puntatore alla porzione di memoria di dimensione `size` oppure `NULL` se si è verificato un errore
- Definita nella standard library (`stdlib.h`)
- Anche funzione `calloc`

```
void *calloc(size_t count, size_t size);
```

  - Oltre ad allocare la quantità di memoria richiesta in modo contiguo, la inizializza con byte pari a 0

malloc(): esempio

```
1) int *p;
2) p = malloc(sizeof(int));
3) *p = 7;
4) printf("%d", *p);
```

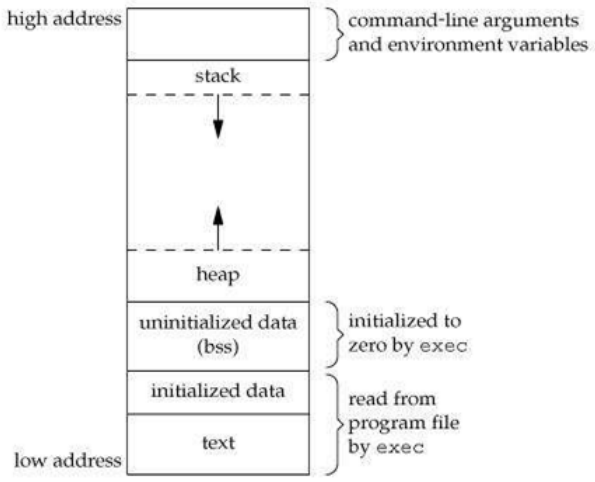
Spiegazione

1) Definisce una variabile di tipo puntatore a int
2) Crea in memoria una variabile di tipo int e restituisce in p l'indirizzo della variabile creata
3) Alla variabile di tipo int creata si assegna il valore 7
4) Viene stampato 7

- Attenzione, manca il controllo sulla malloc: vedi slide successiva

Layout della memoria: stack e heap

- Le variabili locali di ogni funzione sono nello *stack*
- Le zone di memoria allocate con malloc sono in una zona diversa, detta *heap*



Allocazione dinamica di memoria

- L'allocazione dinamica della memoria avviene nello heap
- Se lo spazio di memoria allocabile dinamicamente è esaurito viene restituito il puntatore `NULL`: occorre controllare!
- Esempio:

```
int *pt;
pt = malloc(sizeof(int)); // allocazione dinamica
if (pt == NULL) {
 printf("allocazione fallita\n");
 exit(EXIT_FAILURE);
}
```

  - In caso di mancanza di memoria il programma termina con `exit(EXIT_FAILURE);` che notifica il fallimento
    - ✓ `EXIT_FAILURE` definita in stdlib.h

Funzione free()

- La funzione free permette di deallocare la memoria allocata dinamicamente

```
void free(void* ptr);
```

- `free` dealloca lo spazio di memoria puntato da `ptr`, rendendolo disponibile per usi futuri
- Definita nella standard library
- Attenzione: per comparire in una `free` il puntatore `ptr` deve essere stato usato in una precedente chiamata a `malloc`

Esempio: allocazione dinamica di un vettore

Codice sorgente: mem_alloc.c

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  /* Alloca dinamicamente la memoria per un array di tipo int */
  int *pa, dim;

  printf("Inserire la dimensione dell'array da allocare: ");
  scanf("%d", &dim);
  pa = (int *)malloc(dim * sizeof (int)); //casting, malloc ritorna void *
  //pa = (int *)calloc(dim, sizeof (int));
  if (pa == NULL) { /* La memoria non può essere allocata */
    printf("Allocazione memoria fallita\n");
    exit(EXIT_FAILURE);
  }
  else { /* Allocazione avvenuta con successo */
    /* ... */
    free(pa); /* Dealloca la memoria */
    pa = NULL; /* Il puntatore non può essere più usato fino ad un
                  riassegnamento effettuato con malloc/calloc */
  }
  return 0;
}
```


Tempi di vita delle variabili allocate dinamicamente

- Per le variabili allocate dinamicamente il tempo di vita corrisponde al periodo che va
  - dal momento in cui la variabile viene allocata
  - al momento in cui la variabile viene deallocata
- Il tempo di vita di una variabile allocata dinamicamente è definito solo a tempo di esecuzione
- Se una variabile allocata dinamicamente non è deallocata esplicitamente, il suo tempo di vita si prolunga fino alla terminazione del programma
- Può verificarsi il caso in cui una zona di memoria allocata non sia più accessibile, se non ci sono variabili che la "riferiscono"

Problemi nella gestione della memoria: Memory leak

- *Attenzione:*
  - La memoria nello stack è deallocata automaticamente
  - La memoria nell'heap **non** è deallocata automaticamente
- Fare attenzione al problema del *memory leak*
  - Consumo non voluto di memoria dovuto alla mancata deallocazione dalla stessa di variabili/dati non più utilizzati
- Esempio:

```
int *temp, k;
for (k= 1; 1; k++) {
  printf("k  = %d\n", k);
  temp= malloc(sizeof(int)); *temp = k;
}
```

- Tool utile: valgrind http://valgrind.org

Problemi nella gestione della memoria: Stack overflow

- Stack overflow avviene quando è richiesto l'uso di una quantità troppo elevata di memoria nello stack
- Lo stack delle chiamate contiene una quantità limitata di memoria, fissata di solito all'avvio del programma
  - La dimensione dello stack dipende da molteplici fattori: linguaggio di programmazione, architettura della macchina, disponibilità di memoria nel sistema, …
  - Se viene usata troppa memoria nello stack, avviene un overflow e si verifica un crash del programma
  - Questa classe di bug è di solito causata da uno dei due tipi di errori di programmazione
    - ✓ Ricorsione infinita: vedere esempio in stack_overflow.c
    - ✓ Uso di variabili molto grandi

Codice sorgente: stack_overflow.c

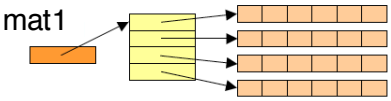
Allocazione dinamica di matrici

- Esaminiamo 4 possibili modi per allocare dinamicamente un array bidimensionale (matrice)
- Differenze
  - Matrice come blocco di memoria contiguo oppure non continuo
  - Modalità di accesso agli elementi della matrice
  - Come passare la matrice allocata dinamicamente come parametro di ingresso di una funzione

Allocazione dinamica di matrici: versione 1

- Con array di puntatori a tipo (ad es. `int`), blocco di memoria non contiguo:

```
//nr: numero di righe, nc: numero di colonne
int **mat1;
mat1 = (int **)malloc(nr*sizeof(int *));
for (i=0; i<nc; i++)
    mat1[i] = (int *)malloc(nc*sizeof(int));
```



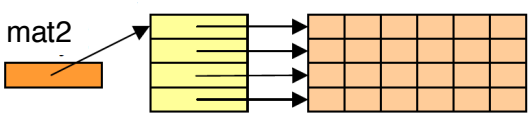
- malloc crea un array di nr puntatori a `int` e per ogni puntatore alloca un array di `int` di lunghezza nc
- Come si accede agli elementi della matrice:

```
mat1[riga][colonna] = 5;
```

- nr e nc possono essere variabili

Allocazione dinamica di matrici: versione 2

- Con array di puntatori a tipo (ad es. `int`), blocco di memoria contiguo:




```
int **mat2;
mat2 = (int **)malloc(nr*sizeof(int *));
mat2[0] = (int *)malloc(nr*nc*sizeof(int));
for (i=1; i<nr; i++)
    mat2[i] = mat2[0] + i*nc;
```

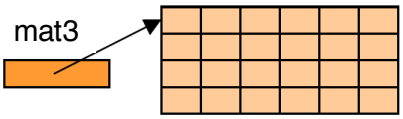
- malloc crea un array di `nr` puntatori a `int`, alloca un blocco per tutta la matrice, calcola e assegna ad ogni puntatore l'indirizzo di ciascuna riga
- Come si accede agli elementi della matrice:

```
mat2[riga][colonna] = 5;
```

- `nr` e `nc` possono essere variabili

Allocazione dinamica di matrici: versione 3

- Con array di tipo (ad es. `int`), blocco di memoria contiguo ma la matrice è simulata con un array monodimensionale:




```
int *mat3;
mat3 = (int *)malloc(nr*nc*sizeof(int));
```

- malloc crea un blocco per tutta la matrice e accede agli elementi calcolandone la posizione (offset) riferita al primo elemento
  - La matrice è in realtà un vettore
- Accesso agli elementi della matrice:

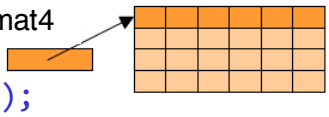
```
mat3[riga*nc+colonna] = 5;
```

- `nr` e `nc` possono essere variabili

Allocazione dinamica di matrici: versione 4

- Con puntatore ad array di tipo (ad es. `int`), blocco di memoria contiguo:

```
int (*mat4)[NCOL];
mat4 = (int (*)[NCOL])malloc(nr*sizeof(*mat4));
```



- malloc crea un blocco per tutta la matrice e lo fa puntare da un puntatore
  - Il tipo di `mat4` è "puntatore a array di `NCOL int`"
  - `*mat4` è un "vettore di `NCOL int`"
  - La sua dimensione è quella di una riga della matrice (`NCOL`)
- Accesso agli elementi della matrice:

```
mat4[riga][colonna] = 5;
```

- `nr` può essere variabile, `NCOL` è una costante

Passaggio di una matrice come parametro

- Esaminiamo il passaggio di un array (vettore o matrice) come parametro di input
  - Avviene simulando il passaggio per riferimento (si passa il puntatore)

Codice sorgente: pass_array.c

Allocazione di matrice

- Esaminiamo con un esempio le varie modalità di allocazione di una matrice
  - Esercizio 4 della prova in itinere 2018/19

Codice sorgente: itinere1819.c