Laboratorio II

Corso A

Lezione 4

Aritmetica dei puntatori

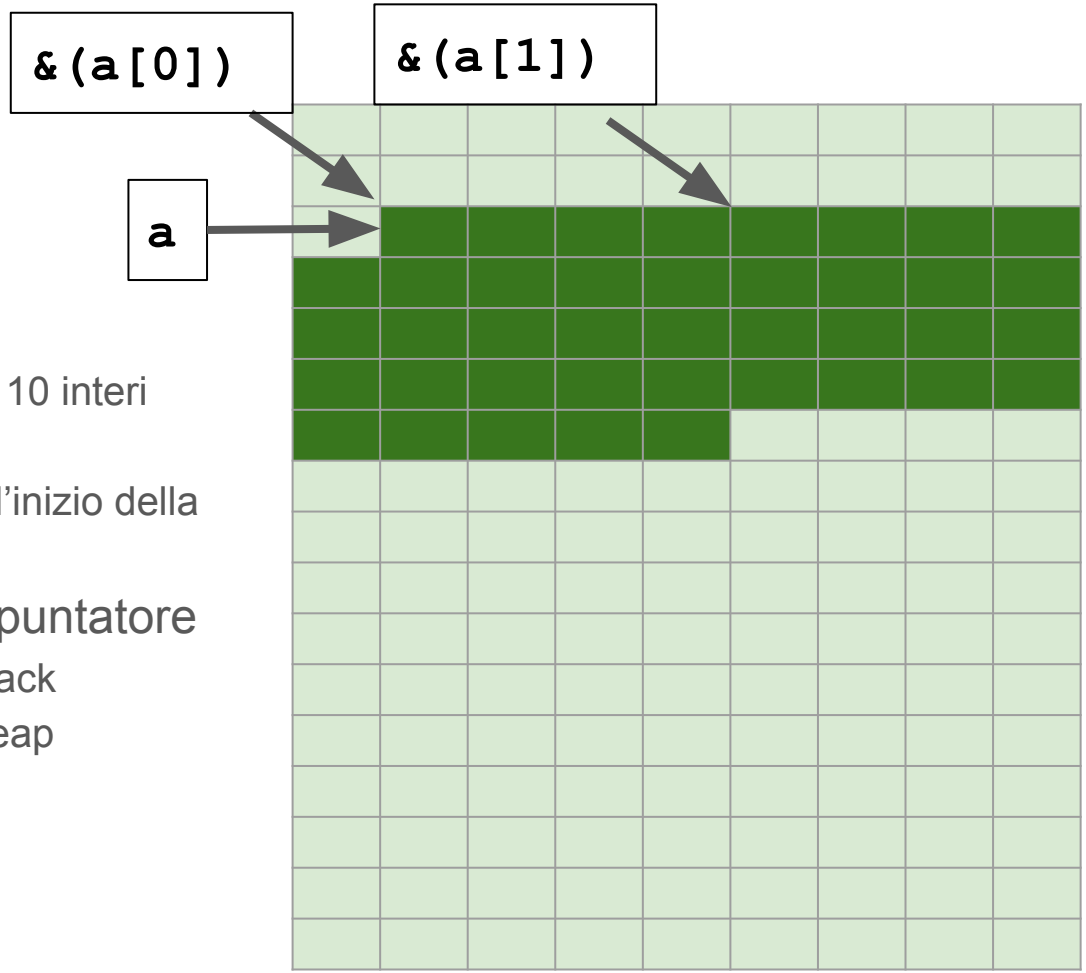
Allocazione dinamica della memoria

Array di puntatori

Liste concatenate

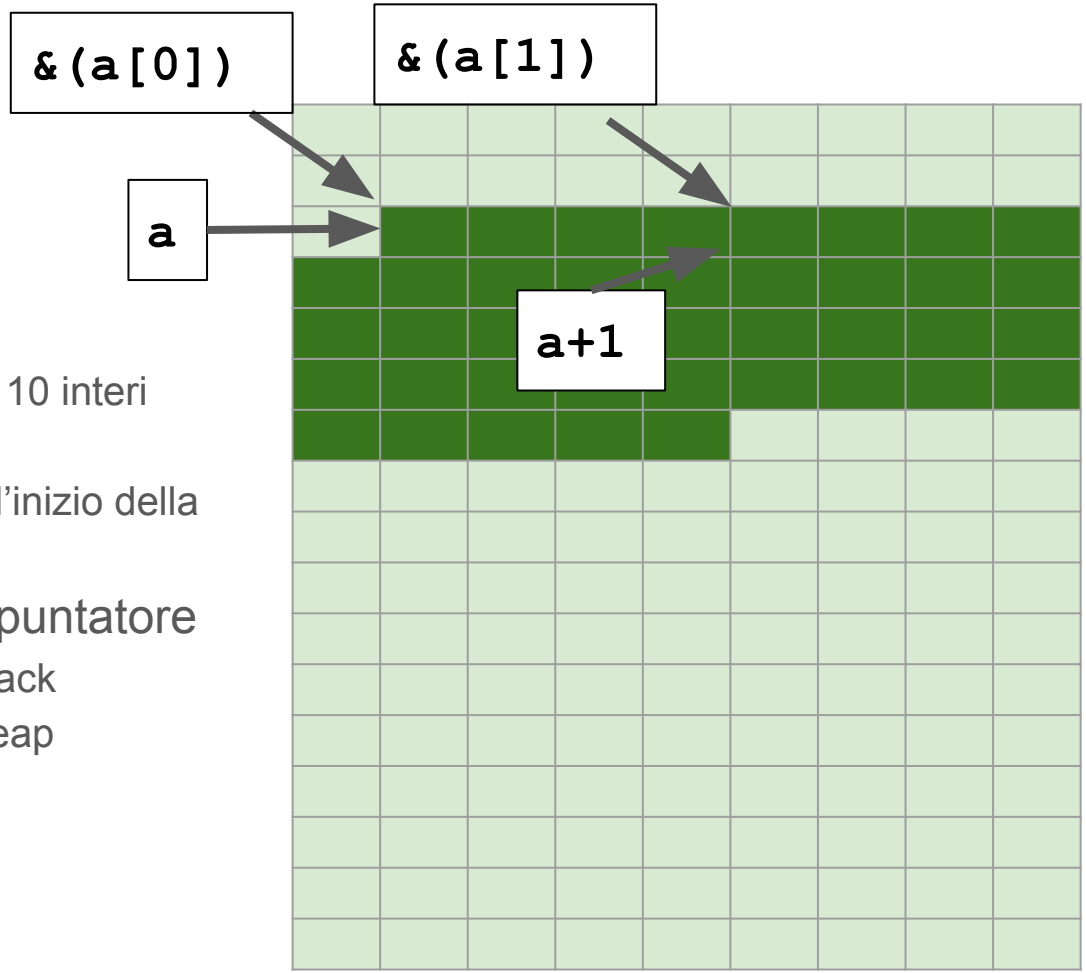
Array e puntatori

- `int a[10];`
  - crea una variabile `a`
  - alloca in memoria sullo stack 10 interi contigui
  - memorizza in a l'indirizzo dell'inizio della zona di memoria allocata
- La variabile `a` è in realtà un puntatore
  - array statici - allocati sullo stack
  - array dinamici - allocati sul heap
- a == &(a[0])
- &(a[1]) == ????



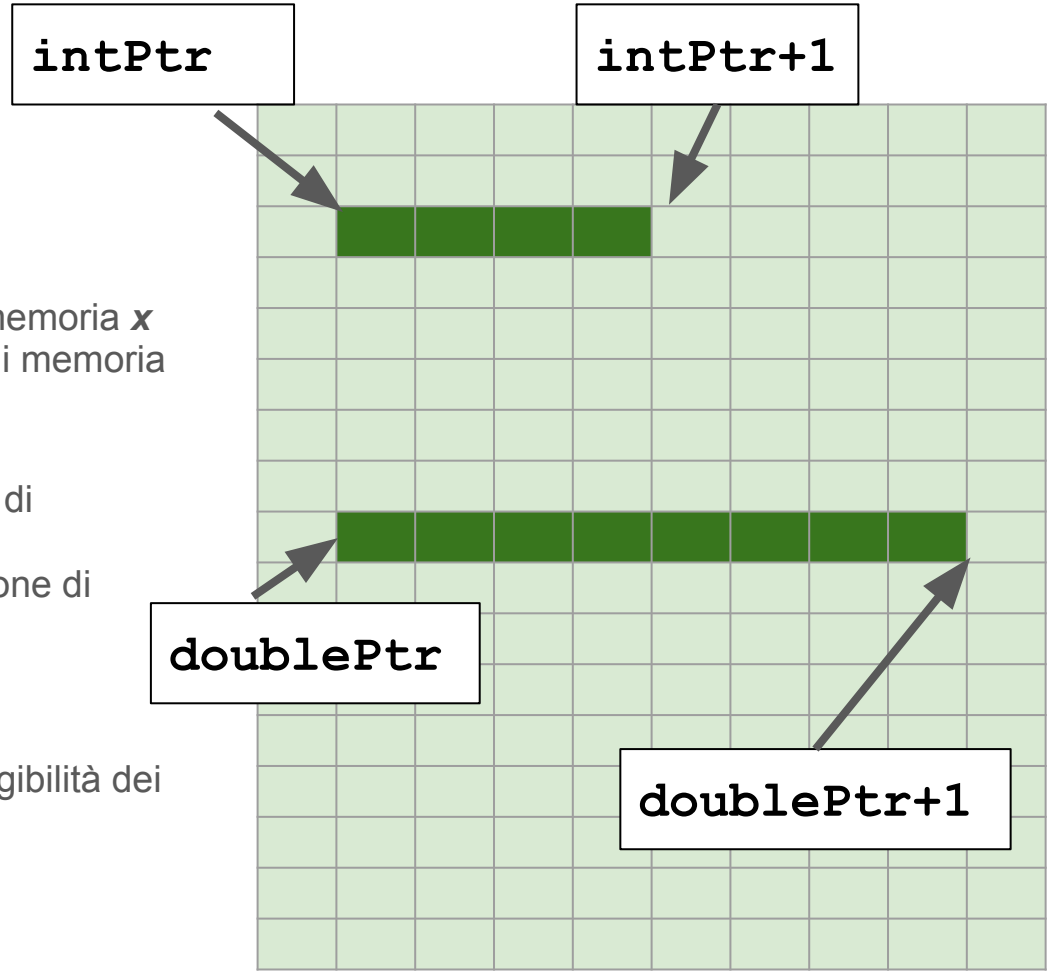
Array e puntatori

- `int a[10];`
  - crea una variabile `a`
  - alloca in memoria sullo stack 10 interi contigui
  - memorizza in a l'indirizzo dell'inizio della zona di memoria allocata
- La variabile `a` è in realtà un puntatore
  - array statici - allocati sullo stack
  - array dinamici - allocati sul heap
- a == &(a[0])
- &(a[1]) == (a+1)
- &(a[i]) == (a+i)



Aritmetica dei puntatori

- `int* intPtr = &a;`
  - `intPtr` punta alla locazione di memoria ***x***
  - `intPtr+i` punta alla locazione di memoria ***x+i*sizeof(int)***
- `double* doublePtr=&a;`
  - `doublePtr` punta alla locazione di memoria ***x***
  - `doublePtr+i` punta alla locazione di memoria ***x+i*sizeof(double)***
- etc…
- `int a[10];`
  - `a[i]` - sintassi che facilita la leggibilità dei programmi
  - `a[i]` - trasformato in `*(a+i)`



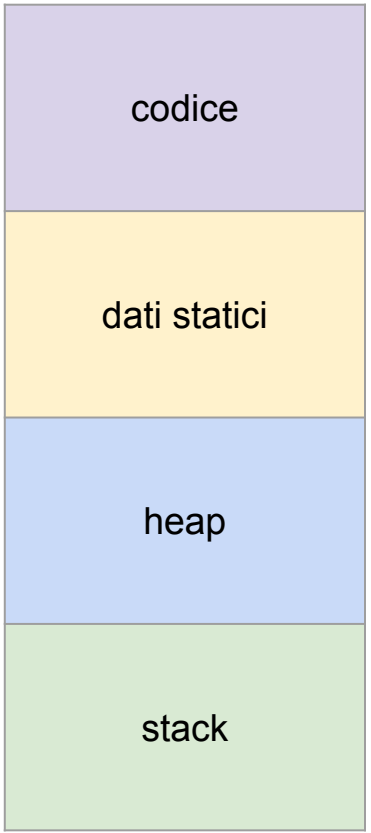
Bubble sort usando aritmetica dei puntatori

```c
int main(){
  float v[N];
  int dim=0, i;
  int swapped=1;

  do{
    scanf("%f", &v[dim]);
    dim++;
  } while(v[dim-1]!=0);

  while(swapped){
    swapped=0;
    for(i=0;i<dim-1;i++)
      if (v[i]<v[i+1]){
        float aux=v[i];
        v[i]=v[i+1];
        v[i+1]=aux;
        swapped=1;
      }
  }

  for(i=0;i<dim;i++)
    printf("%.2f ", v[i]);

  return 0;
}
```

```c
int main(){
  float v[N];
  int dim=0, i;
  int swapped=1;

  do{
    scanf("%f", v+dim);
    dim++;
  } while(*(v+dim-1)!=0);

  while(swapped){
    swapped=0;
    for(i=0;i<dim-1;i++)
      if (*(v+i)<*(v+i+1)){
        float aux=*(v+i);
        *(v+i)=*(v+i+1);
        *(v+i+1)=aux;
        swapped=1;
      }
  }

  for(i=0;i<dim;i++)
    printf("%.2f ", *(v+i));

  return 0;
}
```

Allocazione della memoria

- Programma e dati - memoria
- Zona dati
  - Dati statici - possono essere inferiti a tempo di compilazione
    - Variabili globali, statici
  - Stack - record di attivazione
    - 1 per ogni blocco di codice (incluso chiamate di funzioni)
    - spazio per variabili locali
  - Heap - dati dinamici (runtime), frammentati
    - Memoria allocata dinamicamente - tramite puntatori



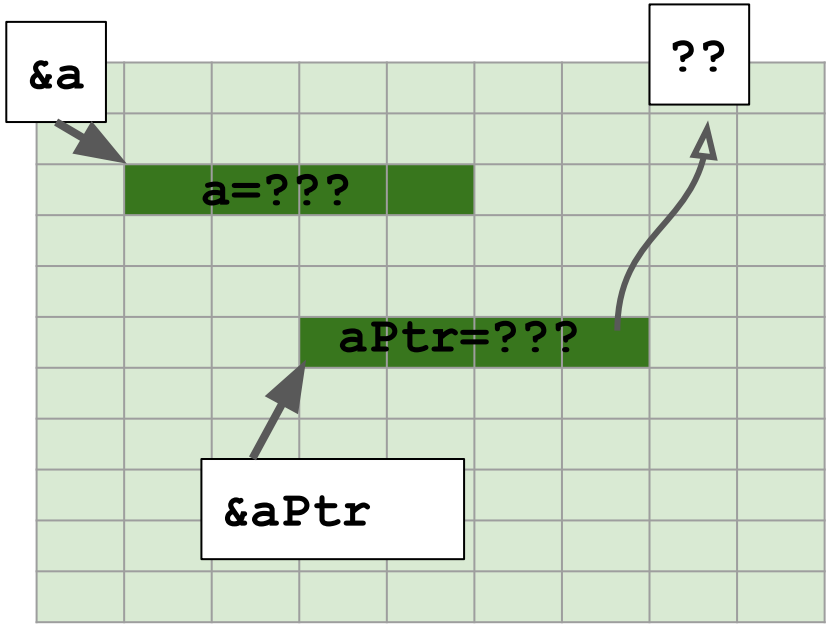
Allocazione della memoria

`int a;`

- Alloca spazio per un intero (`sizeof(int)`)
- Qual'è il valore?

`int * aPtr;`

- Alloca spazio per un puntatore. (`sizeof (int*)`)
- Qual'è il valore (indirizzo di memoria a cui punta)?



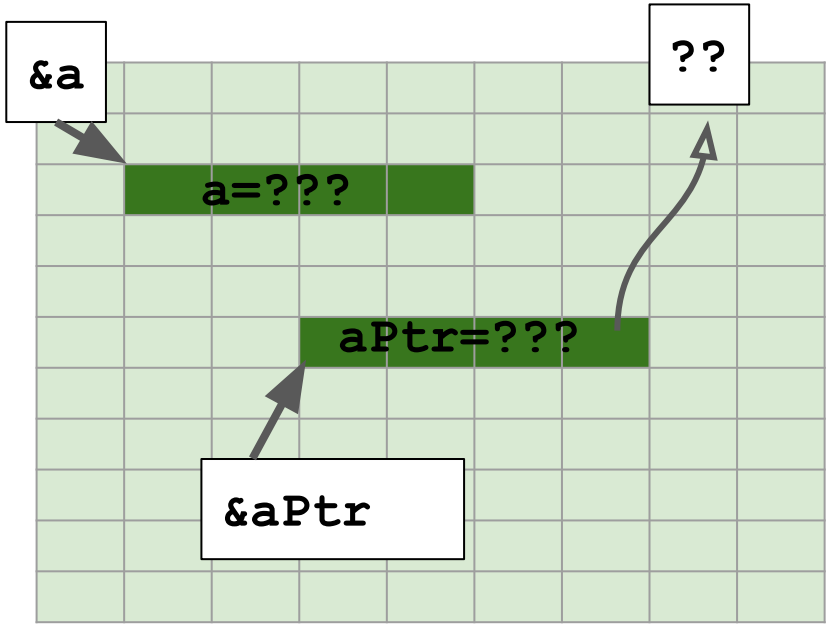
Allocazione della memoria

```
int a;
```

- Alloca spazio per un intero (`sizeof(int)`)
- Qual'è il valore? - non definito

```
int * aPtr;
```

- Alloca spazio per un puntatore. (`sizeof (int*)`)
- Qual'è il valore (indirizzo di memoria a cui punta)? - non definito



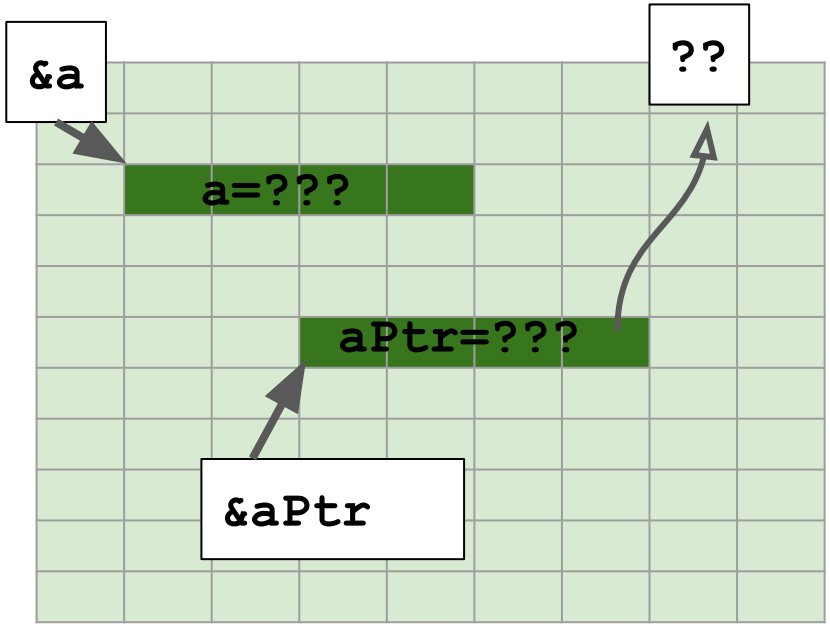
Allocazione della memoria

```
int a;
```

- Alloca spazio per un intero (`sizeof(int)`)
- Qual'è il valore? - non definito

```
int * aPtr;
```

- Alloca spazio per un puntatore. (`sizeof (int*)`)
- Qual'è il valore (indirizzo di memoria a cui punta)? - non definito
- Cosa c'è all'indirizzo di memoria a cui punta (`*aPtr`)? - non definito



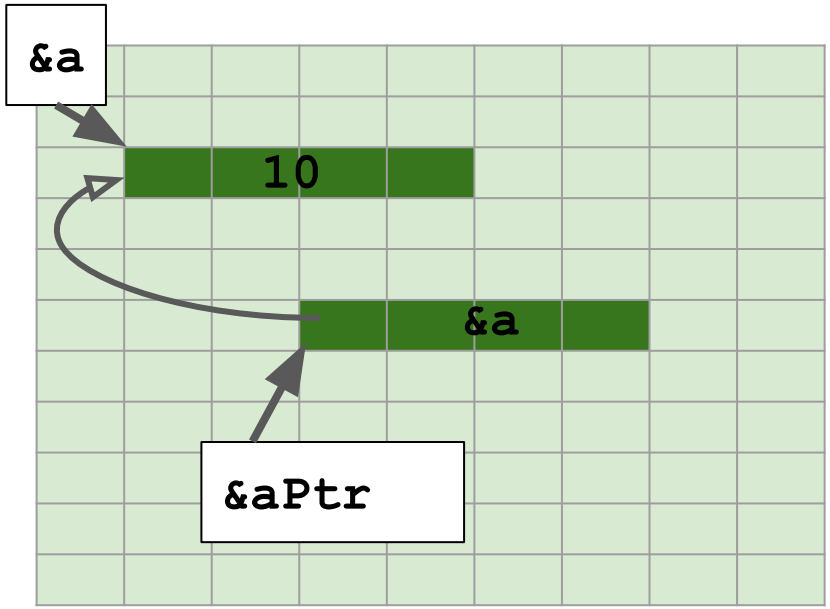
Allocazione statica

```
int a;
```

- Alloca spazio per un intero (`sizeof(int)`)
- Qual'è il valore? - `a=10;`

```
int * aPtr;
```

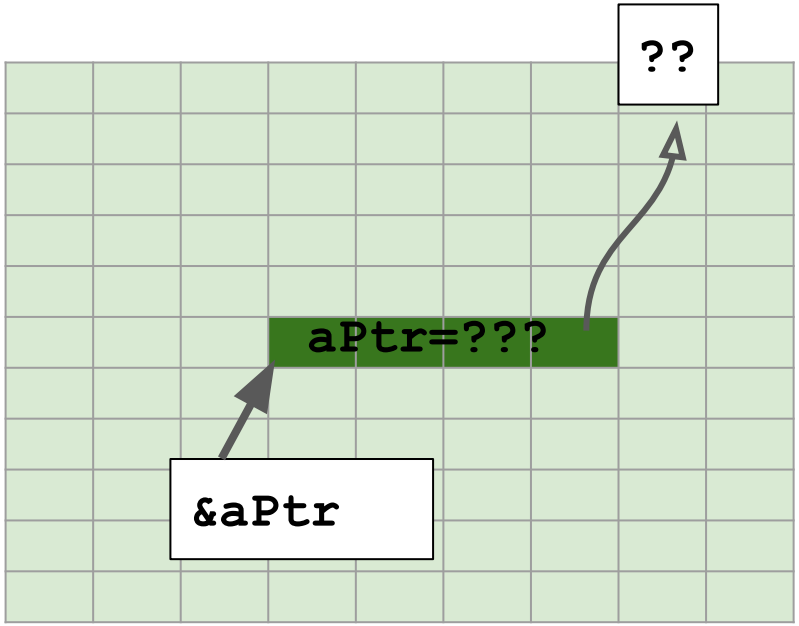
- Alloca spazio per un puntatore. (`sizeof (int*)`)
- Qual'è il valore (indirizzo di memoria a cui punta)? - `aPtr=&a;`
- Cosa c'è all'indirizzo di memoria a cui punta (*aPtr)? - `10`



Allocazione dinamica

```
int * aPtr;
```

- Alloca spazio per un puntatore. (`sizeof (int*)`)
- Qual'è il valore (indirizzo di memoria a cui punta)? - non definito
- Cosa c'è all'indirizzo di memoria a cui punta (`*aPtr`)? - non definito

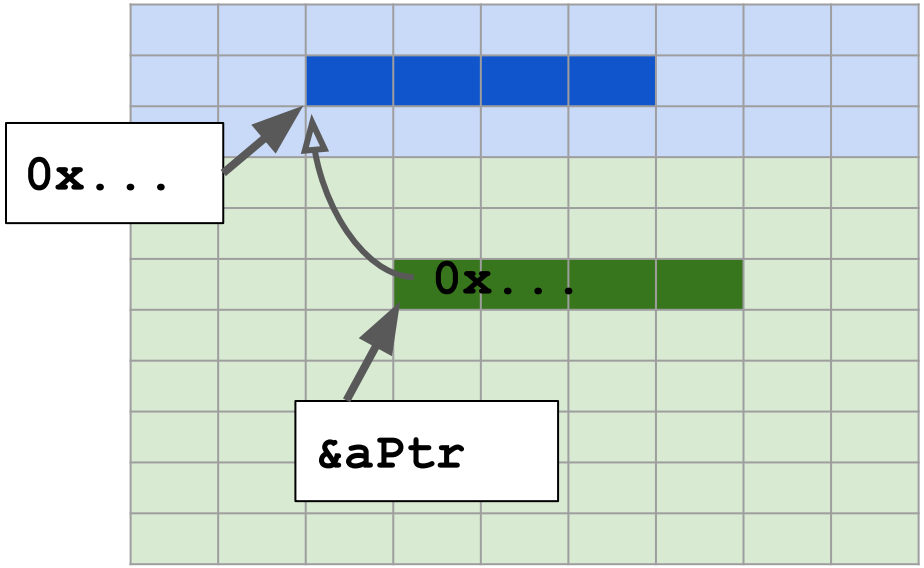


La libreria **`stdlib`** offre funzionalità per allocare memoria in modo dinamico

Allocazione dinamica

```
int * aPtr;
```

- Alloca spazio per un puntatore. (`sizeof (int*)`)
- Qual'è il valore (indirizzo di memoria a cui punta)? - un indirizzo `0x...`
- Cosa c'è all'indirizzo di memoria a cui punta (`*aPtr`)? - non definito

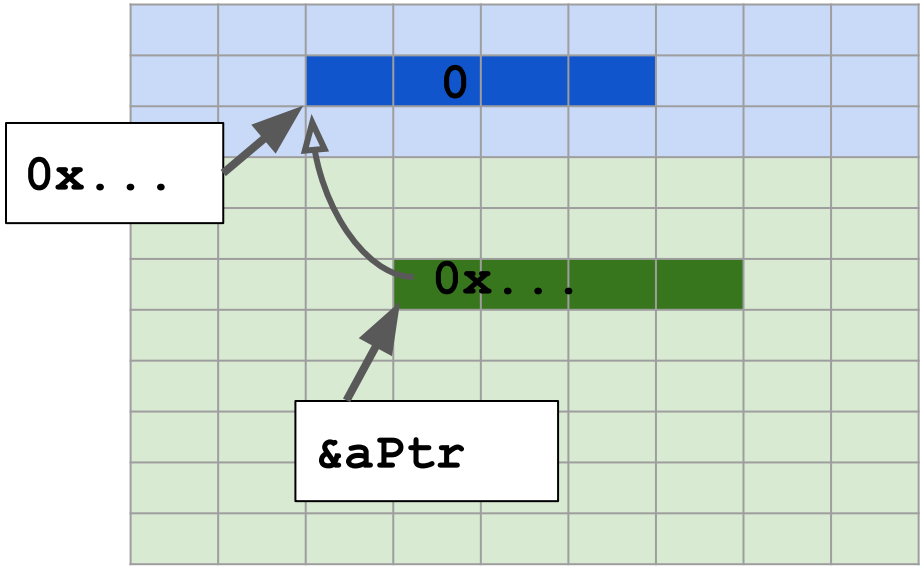



```
aPtr=(int*)malloc(sizeof(int));
```

Allocazione dinamica

```
int * aPtr;
```

- Alloca spazio per un puntatore. (`sizeof (int*)`)
- Qual'è il valore (indirizzo di memoria a cui punta)? - un indirizzo `0x...`
- Cosa c'è all'indirizzo di memoria a cui punta (`*aPtr`)? - il valore 0




```
aPtr=(int*)calloc(1,sizeof(int));
```

`malloc` e `calloc`

- `void *malloc (size_t n);`
  - Alloca `n` byte e restituisce l'indirizzo di memoria dove sono allocati
- `void *calloc (size_t n, size_t size);`
  - Alloca un array di `n` oggetti di dimensione `size`, inizializza ogni posizione con `0` e restituisce l'indirizzo di memoria del primo elemento.
- Valore di ritorno:
  - Restituiscono un puntatore `void*` - simile ad un tipo generico, bisogna fare il casting al tipo giusto.
  - Il valore `NULL` se non riesce ad allocare la memoria - bisogna controllare che il valore sia `!= NULL`

`void*` - puntatore generico. Può essere usato (anche) in prototipi di funzioni, ma per dereferenziare bisogna fare cast ad un tipo concreto.

`NULL` è una costante (`stdlib.h`), di valore `0` - si utilizza con i puntatori

void*

```c
#include <stdio.h>

int main(void) {

  int a=10;
  void* aP=&a;

  printf("%d\n", *aP);

}
```

```
main.c:8:18: error: argument type 'void' is incomplete
  printf("%d\n", *aP);
                 ^
1 error generated.
exit status 1
```

`void*` - puntatore generico. Può essere usato (anche) in prototipi di funzioni, ma per dereferenziare bisogna fare cast ad un tipo concreto.

```c
#include <stdio.h>

int main(void) {

  int a=10;
  void* aP=&a;

  printf("%d\n", *(int*)aP);

}
```

`free`

- Libera memoria allocata con malloc o calloc.
  - `free(aPtr);`
- Bisogna liberare la memoria quando un oggetto non è più necessario.
- Errori comuni:
  - Non liberare la memoria e poi perdere il puntatore - memory leak
    - Di solito dà problemi di performance ma non errori di esecuzione - al massimo out of memory error.
  - Liberare memoria allocata in modo statico
  - Utilizzare un puntatore dopo aver liberato la memoria

Array di puntatori

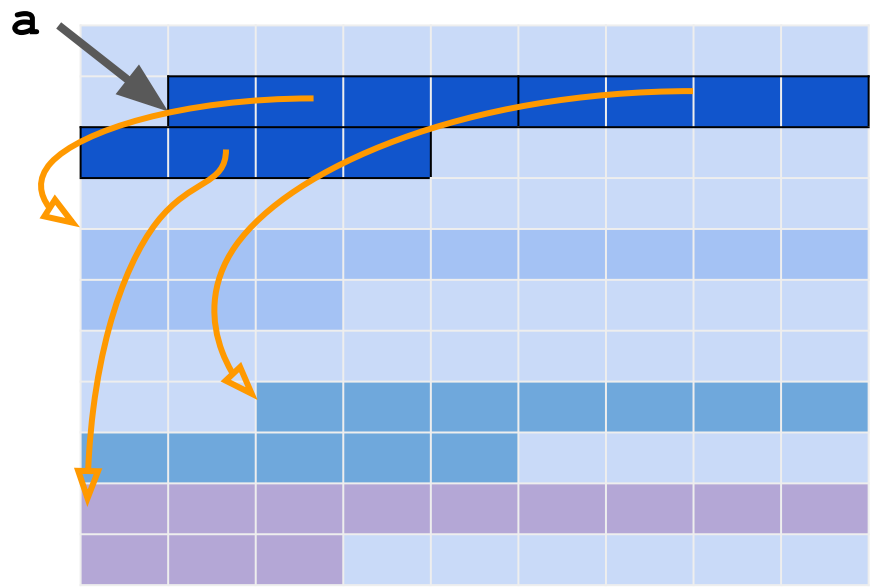
- I puntatori sono variabili, quindi possiamo avere array di puntatori
- `int *ps[10];`
  - alloca memoria per un array di 10 puntatori
  - `ps[i]` -> un puntatore non inizializzato
    - bisogna assegnare un valore a tutti i puntatori
- Possiamo costruire anche array dinamici di puntatori
  - `int** ps = (int**)malloc(n*sizeof(int*))` -> `n` puntatori a `int`, non inizializzati
  - `int** ps = (int**)calloc(n,sizeof(int*))` -> `n` puntatori a `int`, inizializzati a `NULL`

Array di puntatori vs array bidimensionali

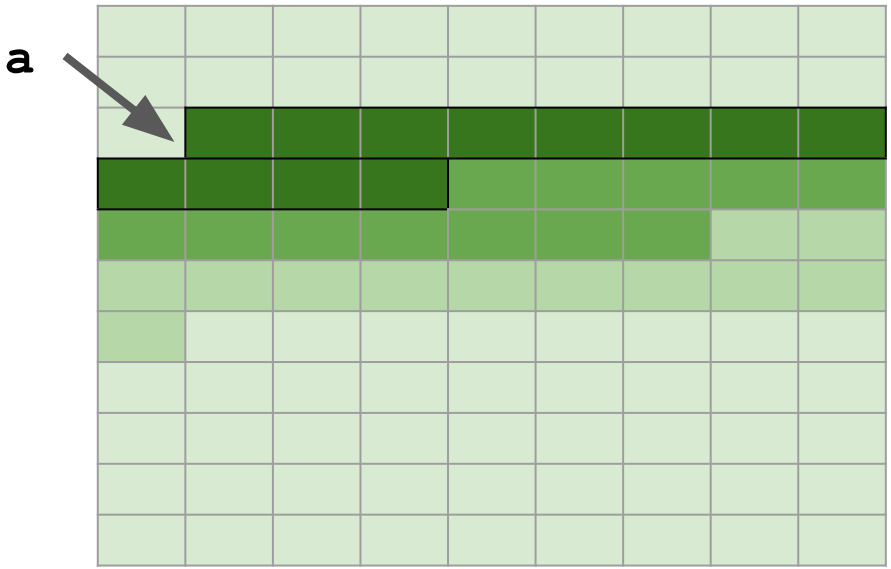
- `int** ps = (int**)malloc(n*sizeof(int*))`
- Se gli puntatori in `ps` rappresentano array, allora `ps` è simile a un array bidimensionale
- Più generico del array definito come `int a[N][M];`
  - Ogni riga può avere dimensione diversa, senza spreco di memoria
- Più facile da passare come parametro a funzioni (non serve conoscere `M`)
- Organizzazione in memoria è diversa

Array di puntatori vs array bidimensionali

```
int** a=(int**)malloc(3*sizeof(int*))
for (int i=0;i<3;i++)
    a[i]=(int*)malloc(3*sizeof(int))
```




```
int a[3][3];
```



Allocazione dinamica vs statica

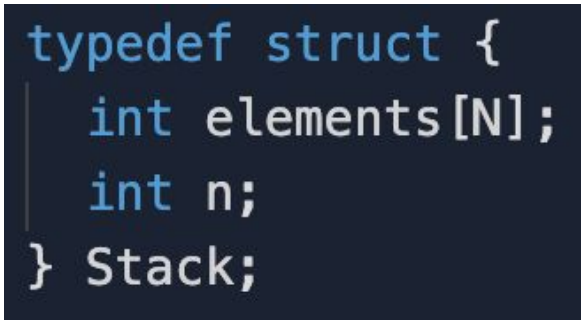
Vantaggi

- Più controllo sulla memoria utilizzata
- Meno spreco di memoria
- Programmi più generici
- Più modularità, passaggio più facile di parametri

Svantaggi

- Complessità aggiunta della gestione della memoria
- Possibilità di memory leak

Puntatori a struct




```
Stack* s=(Stack*)malloc(sizeof(Stack));
```

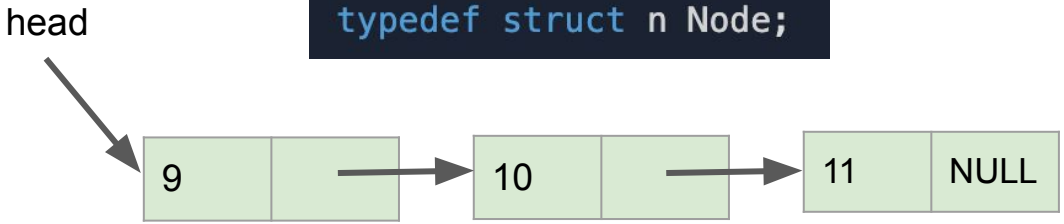
- Alloca memoria per un oggetto `Stack` (un array più un intero)
- Per accedere ai campi utilizzare operatore `->`
- Una `struct` può contenere come membro *puntatori* a se stessa (*self-referential struct*)

Liste concatenate

Posso tenere in ogni elemento un riferimento al prossimo elemento.

Per gestire la lista lavoro con un riferimento al primo elemento.

```
struct n{
  int val;
  struct n * next;
};
typedef struct n Node;
```




```
int main(void) {

  Node* head=NULL; //lista vuota

  Node elem;
  elem.val=10;
  elem.next=NULL;

  head=&elem; //lista con un elemento

  Node elem1; //elemento da aggiungere in coda
  elem1.val=11;
  elem1.next=NULL;
  elem.next=&elem1;//aggiungo in coda

  Node elem2;//elemento da aggiungere in testa
  elem2.val=9;
  elem2.next=head;//aggiungo in testa
  head=&elem2;

  Node* l=head;
  while(l!=NULL){
    printf("%d ",l->val);
    l=l->next;
  }
}
```

Esempi

Funzione di lettura di una matrice di dimensione ***nXm*** conosciuta al runtime

Pila con liste concatenate

Inserimento ordinato in una lista (anche RIC)

Domande?