RIPRENDIAMO I PUNTATORI

- Ogni variabile in C è una astrazione di una cella di memoria a cui corrisponde un nome, un contenuto e un indirizzo.

```
int a = 5;
```

- Esistono in C particolari variabili dette **puntatori** che possono contenere un indirizzo di una variabile

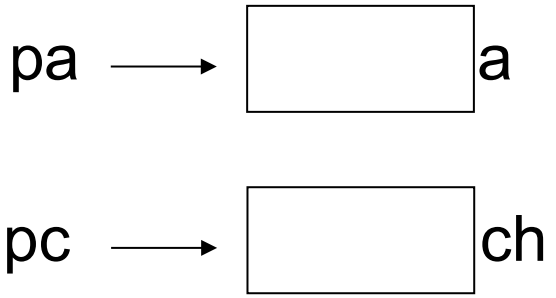
RIPRENDIAMO I PUNTATORI

- La sintassi è

  `tipoBase * varPunt;`

- dove **`varPunt`** è definita come variabile di tipo puntatore a **`tipoBase`**
- Quindi **`varPunt`** può contenere indirizzi di variabili di tipo **`tipoBase`** che può essere int, float, double o char

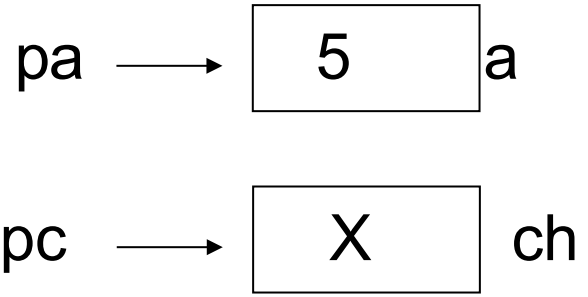
```
int a; char ch;
int *pa; char *pc;
pa = &a;
pc = &ch;
```



RIPRENDIAMO I PUNTATORI

- Ora se assegnamo valori ad a e c

```
int a; char ch;
int *pa; char *pc;
pa = &a;
pc = &ch;
   a = 5;
ch = 'X';
```




```
printf("a=%d  ch =%c", a,ch);
printf("a=%d  ch =%c", *pa,*pc);
```

Hanno lo stesso effetto

ARRAY E PUNTATORI

- Sappiamo che il nome dell'array è l'indirizzo della prima cella di memoria

```
int buf[100];
int *punt;
```

`punt = buf` oppure `punt = &buf[0]`

sono equivalenti.

Da ora in poi possiamo usare punt per accedere all'array. COME?????

ARRAY E PUNTATORI

```
int buf[100];
int *punt;
```

`punt = buf` oppure `punt = &buf[0]`

sono equivalenti.

Da ora in poi possiamo usare punt per accedere all'array.

```
buf[7] = 5;
*(punt + 7) = 5
```

Hanno lo stesso effetto di assegnare 5 all'ottava cella del vettore

ARITMETICA DEI PUNTATORI

- Per una variabile di tipo puntatore esistono operazioni aritmetiche: l'incremento, il decremento, la somma, la sottrazione ecc…
- Ma qual è l'esatto significato di tali operazioni?
- Se ho una variabile di tipo puntatore a carattere

```
char *pc;
```

- e **pc** vale 10 (indirizzo = 10) non è detto che **pc++** valga 11. Tutto dipende dal tipoBase puntato

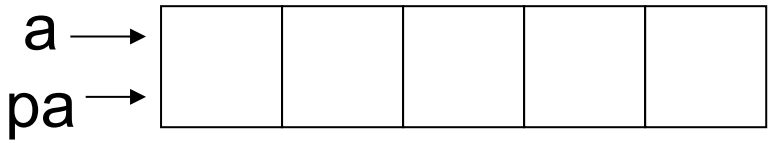
ARITMETICA DEI PUNTATORI

- Incrementare un puntatore di uno significa far saltare il puntatore alla prossima locazione corrispondente all'elemento di memoria il cui tipo corrisponde al tipoBase

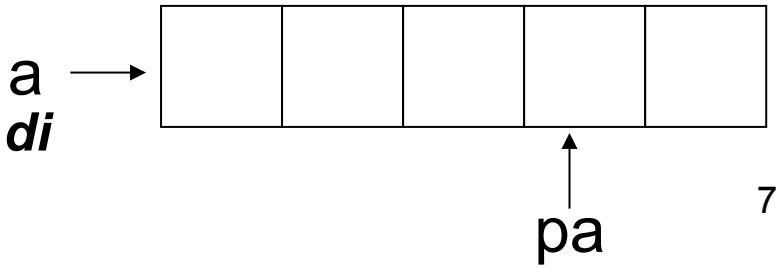
```
int a[5];

int* pa;

pa = a
```




```
pa  = pa + 3;
```



Corrisponde allo spostamento di pa di 3 posizioni dove <u>ogni posizione occupa lo spazio di un int</u>

ALLOCAZIONE STATICA: LIMITI

- Per quanto sappiamo finora, in C le variabili sono sempre **definite *staticamente***
  - **la loro esistenza deve essere prevista e dichiarata a priori**
- I puntatori sono usati nella creazione e manipolazione di variabili ***dinamiche*** create durante l'esecuzione del programma.
  - Tali variabili non hanno un nome esplicito ma vi si accede tramite puntatori

ALLOCAZIONE DINAMICA

Per chiedere nuova memoria "al momento del bisogno" si usa una funzione di libreria che "gira" la richiesta al sistema operativo:

```
void * malloc(int num);
```

La funzione `malloc()`:

- chiede al sistema di allocare **un'area di memoria grande *tanti byte quanti*** ne desideriamo (tutti i byte sono contigui)
- ***restituisce l'indirizzo*** dell'area di **memoria allocata**

LA FUNZIONE `malloc()`

La funzione `malloc(size_t dim)`:

- **chiede al sistema di allocare un'area di memoria grande *`dim`* *byte***
- ***restituisce l'indirizzo* dell'area di memoria allocata** (**`NULL`** se, per qualche motivo, l'allocazione non è stata possibile)
  - è sempre opportuno controllare il risultato di **`malloc()`** prima di usare la memoria fornita
- Il sistema operativo preleva la memoria richiesta **dall'*area heap***

LA FUNZIONE `malloc()`

Praticamente, occorre quindi:

- **specificare quanti byte si vogliono, come parametro passato a `malloc()`**
- ***mettere in un puntatore il risultato fornito da `malloc()` stessa***

Attenzione:

- `malloc()` restituisce **un *puro indirizzo*,** ossia **un puntatore "senza tipo"** `void *`
- per assegnarlo a uno *specifico puntatore* **occorre *un cast esplicito***

ESEMPIO

- Per allocare dinamicamente 12 byte:

```
float *p;
p = (float*) malloc(12);
```

- Per farsi dare *lo spazio necessario per 5 interi* (qualunque sia la rappresentazione usata per gli interi):

```
int *p;
p = (int*) malloc(5*sizeof(int));
```

`sizeof` consente di essere indipendenti dalle scelte dello specifico compilatore/sistema di elaborazione

ALLOCAZIONE DINAMICA DI ARRAY

- **L'esempio precedente può aiutarci a dimensionare array dinamicamente**
- **Finora abbiamo visto che *per variabili di tipo array, occorre specificare a priori le dimensioni (costanti). Questa pratica è particolarmente limitativa***

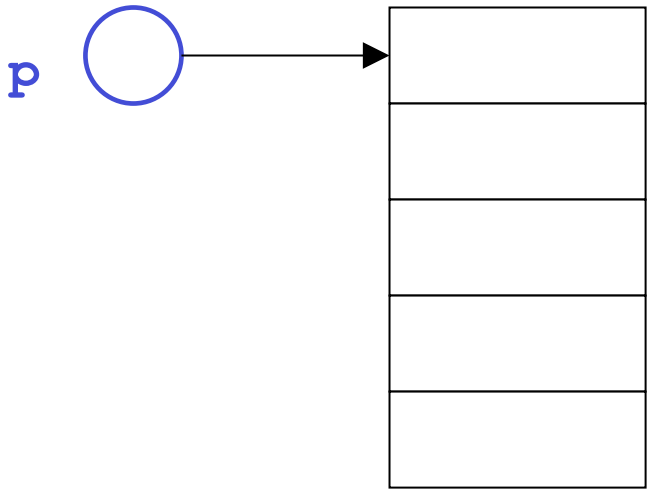
➡ Sarebbe molto utile poter *dimensionare un array "al volo", **dopo aver scoperto quanto grande deve essere***

ESEMPIO

Allocazione:

```
int *p;
p = (int*) malloc(5*sizeof(int));
```

Risultato:



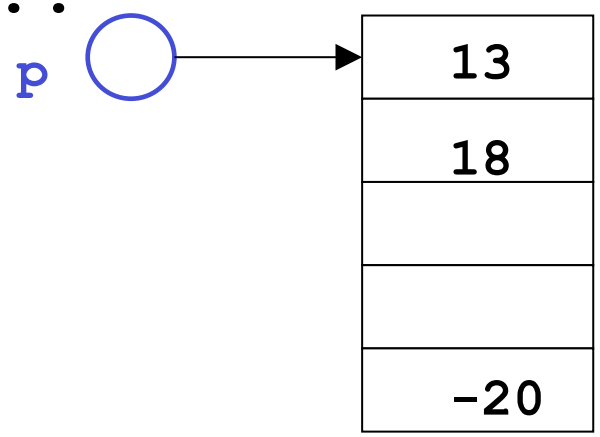
Sono cinque celle contigue, adatte a contenere un int

AREE DINAMICHE: USO

L'area allocata è usabile, in maniera equivalente:

- **o tramite la notazione a puntatore (`*p`)**
- **o tramite la notazione ad array (`[ ]`)**

```
int *p;
p=(int*)malloc(5*sizeof(int));
p[0] = 13; p[1] = 18;...
*(p+4) = -20;
```



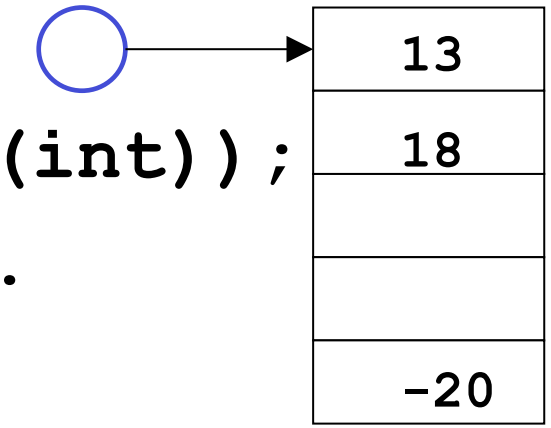
Attenzione a non "eccedere"
l'area allocata dinamicamente.
Non ci può essere alcun controllo

AREE DINAMICHE: USO

Abbiamo costruito un *array dinamico*, le cui dimensioni:

- *non sono determinate a priori*
- *possono essere scelte dal programma in base alle esigenze del momento*
- L'espressione passata a `malloc()` può infatti contenere variabili

```
int *p, n=5;
p=(int*)malloc(n*sizeof(int));
p[0] = 13; p[1] = 18;...
*(p+4) = -20;
```



AREE DINAMICHE: DEALLOCAZIONE

Quando non serve più, l'area allocata deve essere *esplicitamente deallocata*

- ciò segnala al sistema operativo che quell'area è da considerare nuovamente disponibile per altri usi

La deallocazione si effettua mediante la *funzione di libreria* `free()`

```
int *p=(int*)malloc(5*sizeof(int));
    ...
free(p);
```

Non è necessario specificare la dimensione del blocco da deallocare, perché *il sistema la conosce già dalla malloc() precedente*

AREE DINAMICHE: TEMPO DI VITA

Tempo di vita di una area dati dinamica *non* è *legato a quello delle funzioni*

- in particolare, non è legato al tempo di vita della funzione che l'ha creata

Quindi, *una area dati dinamica può sopravvivere anche dopo che la funzione che l'ha creata è terminata*

Ciò consente di

- creare un'area dinamica in una funzione...
- ... usarla in un'altra funzione...
- ... e distruggerla in una funzione ancora diversa

ESERCIZIO 1

Creare un array di float **di dimensione specificata dall'utente**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
 float *v; int n;
 printf("Dimensione: ");
 scanf("%d",&n);
 v = (float*) malloc(n*sizeof(float));
   ... uso dell'array ...
 free(v);
}
```

malloc() e free() sono dichiarate in **stdlib.h**

ESERCIZIO 2

Scrivere una funzione che, dato un intero, **allochi e restituisca una stringa di caratteri della dimensione specificata**

```
#include <stdlib.h>
char* alloca(int n){
  return (char*) malloc(n*sizeof(char));
}
```

NOTA: dentro alla funzione *non* deve comparire la **`free()`**, in quanto scopo della funzione è proprio ***creare un array che sopravviva alla funzione stessa***

ESERCIZIO 2 - CONTROESEMPIO

Scrivere una funzione che, dato un intero, **allochi e restituisca una stringa di caratteri della dimensione specificata**

Che cosa invece non si può fare in C:

```
#include <stdlib.h>
char* alloca(int n){
   char v[n];
   return v;
}
```

ARRAY DINAMICI

- **Un array ottenuto per allocazione dinamica è "dinamico" poiché *le sue dimensioni possono essere decise al momento della creazione*, e non per forza a priori**
- ***Non significa che l'array possa essere "espanso" secondo necessità:*** una volta allocato, l'array ha dimensione *fissa*
- **Strutture dati espandibili dinamicamente secondo necessità esistono, ma *non sono array* (*liste, pile, code, …*)**

DEALLOCAZIONE - NOTE

- Il modello di gestione della memoria dinamica del C richiede che ***l'utente si faccia esplicitamente carico*** anche della ***deallocazione della memoria***
- ***È un approccio pericoloso:*** **molti errori sono causati proprio da un'errata deallocazione**
  - rischio di puntatori che puntano ad aree di memoria ***non più esistenti*** → ***dangling reference***
- **Altri linguaggi gestiscono automaticamente la deallocazione tramite *garbage collector***