POLITECNICO
MILANO 1863

Informatica A

Puntatori

POLITECNICO
MILANO 1863

Variabili

- Finora l'accesso alle variabili è avvenuto soprattutto tramite il nome (identificatore):

```
v = a;
```

- Il valore contenuto nella cella identificata da a è memorizzato nella cella identificata da v
  - È possibile che le variabili occupino più celle

- Ma che cos'altro identifica una variabile?
  - Più volte si è parlato di indirizzi
    - Assembler, scanf, operatore &, array, …

POLITECNICO
MILANO 1863

Indirizzi e valori

- Ogni variabile ha, tra gli elementi che la caratterizzano
  - *Indirizzo*: l'indirizzo della locazione di memoria associata alla variabile
    - Indirizzo della prima cella
  - *Valore*: il valore contenuto nella locazione di memoria associata alla variabile

- L'indirizzo è *immutabile*, mentre il valore *muta* durante l'esecuzione del programma
  - La variabile cambia valore ma non si sposta

Indirizzi e valori: gli assegnamenti

- Quando una variabile è a sinistra di un assegnamento si usa *l'indirizzo* per andarne a modificare il valore

- Quando è a destra, si usa *il valore*
  - A cui accediamo tramite l'indirizzo
  - Gli identificatori servono al programmatore per distinguere agevolmente le variabili …
  - … ma per accedere alla RAM l'esecutore usa (ovviamente) gli indirizzi

```
v=a;
```

Indirizzi

- In molti linguaggi di programmazione il programmatore non può conoscere gli indirizzi delle variabili

- In C è possible conoscere l'indirizzo delle locazioni di memoria associate alle variabili
  - Mediante l'operatore &

L'operatore &

```
int main() {
    int x = 3;
    printf("Indirizzo di x: %p\n", &x);
    printf("Valore di x: %d\n", x);
}
```

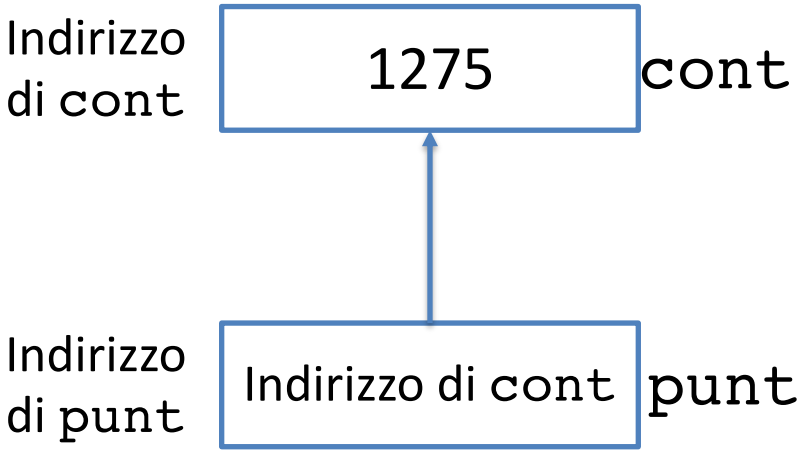
Output del programma

```
Indirizzo di x: 0xbffff984
Valore di x: 3
```

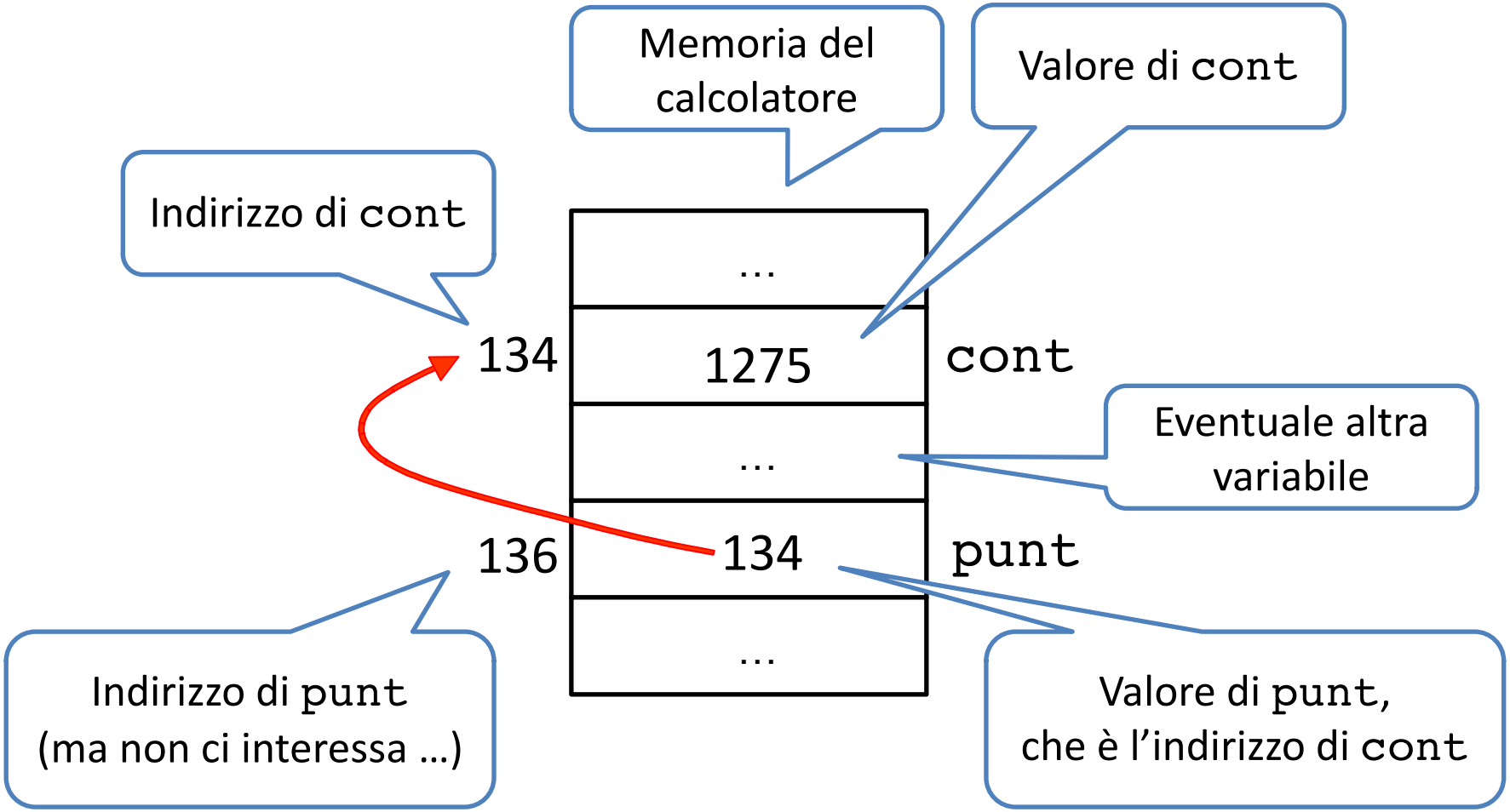
I puntatori

- Le variabili puntatore sono variabili il cui valore è l'indirizzo di un'altra variabile

```
int cont = 1275;
int * punt;
punt = &cont;
```



O anche …


Memoria del calcolatore
Valore di cont
Indirizzo di cont
…
134
1275
cont
…
Eventuale altra variabile
136
134
punt
…
Indirizzo di punt (ma non ci interessa …)
Valore di punt, che è l'indirizzo di cont

Dichiarare i puntatori

```
typedef Tipo *TipoRef;
```

- TipoRef è un puntatore a dati di tipo Tipo

```
typedef int *intRef;
intRef myRef, yourRef;
int * herRef;     /* abbreviazione */
```

- *Attenzione*: puntatori a dati di tipo diverso sono variabili di tipo diverso!

- Suggerimento: usare *Ref* (o *Punt*) in coda al nome per denotare i puntatori

Il modello


```
myRef = &x;
```

- Diciamo che `myRef` *punta a* `x`
- Come possiamo accedere al valore di questa variabile usando `myRef`
  - Dereferenziazione: `*myRef`

Dereferenziazione

- L'operatore unario * è detto di dereferenziazione
- Permette di estrarre il valore della variabile puntata dal puntatore che è argomento dell'operatore

```
typedef int * punt_a_int;
```

- Si legge anche dicendo che dereferenziando un punt_a_int si ottiene un int

```
int x  = 3;
int * p = &x;  /* inizializzazione di p */
printf("il valore di x e' %d\n", *p);
```

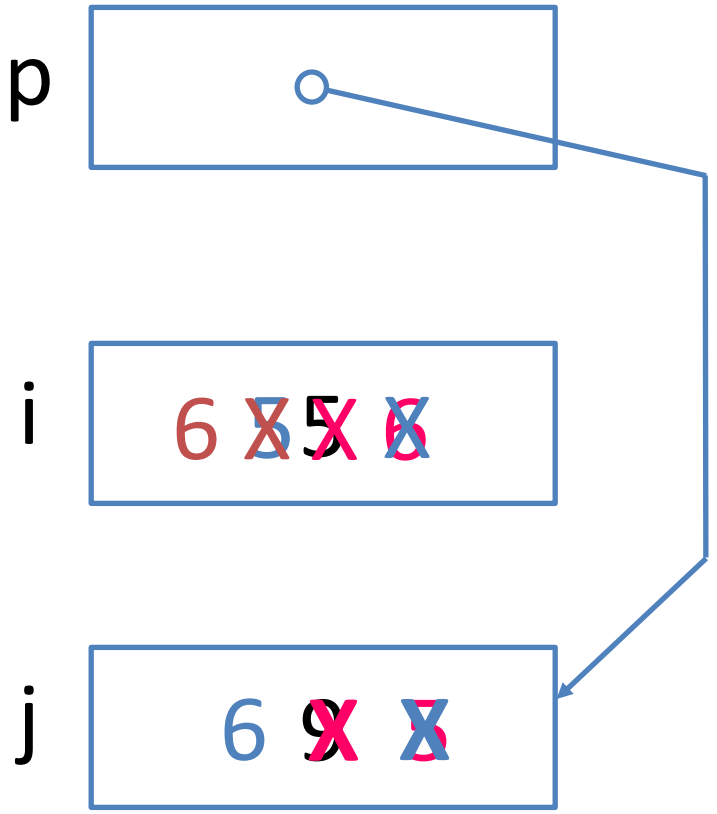
Esercizio: simulazione di esecuzione

```
typedef int * punt;

punt p;
int i = 5, j = 9;
p = &j;

*p = i;
++i;

i = *p;
(*p)++;
p = &i;
*p = j;
```



POLITECNICO
MILANO 1863

Puntatori: riassunto

- int * è la dichiarazione di un puntatore p a un intero

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

- Con &i si denota l'indirizzo della variabile i
- & è l'operatore che restituisce l'indirizzo di una variabile

```
p = &i;
```
(operatore di referenziazione)

- L'operatore opposto è *, che restituisce il valore puntato

```
i = *p;
```
(operatore di dereferenziazione)

- *Attenzione*: non si confondano i molteplici usi dell'asterisco
  - Moltiplicazione, dichiarazione di puntatore, dereferenziazione

Esercizio

- Spiegare la differenza tra

```
p = q;
```

e

```
*p = *q;
```

- Nel primo caso si impone che il puntatore p punti alla stessa variabile a cui punta q

- Nel secondo caso si assegna il valore della variabile puntata da q al valore della variabile puntata da p

Il valore NULL

- NULL: costante simbolica che rappresenta un valore speciale che può essere assegnato a un puntatore
- Significa che la variabile non punta a niente
  - È un errore dereferenziare la variabile

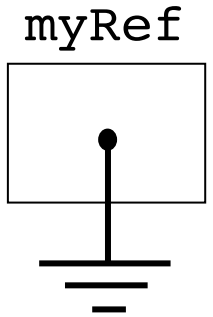
```
myRef = NULL;
```

- Standard ANSI impone che NULL rappresenti il valore 0
  - Test di nullità di un puntatore

```
if (p == NULL)
if (!p)
```

  - Test di non nullità

```
if (p != NULL)
if (p)
```



- Utilizziamo il simbolo della “messa a terra”

Inizializzazione dei puntatori

- Il valore iniziale di un puntatore dovrebbe essere la costante NULL
- NULL significa che non ci si riferisce ad alcuna cella di memoria
- Dereferenziando NULL si ha un errore in esecuzione
- Come al solito, non bisogna *mai* fare affidamento sulle inizializzazioni implicite che il C potrebbe fare
  - Solo alcune implementazioni inizializzano implicitamente a NULL

Puntatori e tipo delle variabili puntate

- Il compilatore segnala l'uso di puntatori a dati di tipo diverso da quello a cui dovrebbero puntare
  - In forma di warning: sono errori potenziali

- I tipi "puntatore a tipo x" e "puntatore a tipo y" sono diversi tra loro

- Il tipo `void *` è compatibile con i puntatori a tutti i tipi

Struct e puntatori

```
typedef struct {
    int     PrimoCampo;
    char    SecondoCampo;
} TipoDato;
TipoDato t;
TipoDato * p = &t;

p->PrimoCampo   = 12;
(*p).PrimoCampo = 12;
```

Equivalenti

Sintassi per accedere ai campi di una struct tramite un puntatore p

Esempio: assegna a due puntatori l'indirizzo degli elementi con valore minimo e massimo in un array

```
#define  SIZE 100

int main() {
    int i,  ArrayDiInt[SIZE];
    int *PuntaAMinore, *PuntaAMaggiore;
    …
    PuntaAMinore = &ArrayDiInt[0];
    for (i=1; i < SIZE; i++)
       if (ArrayDiInt[i] < *PuntaAMinore)
          PuntaAMinore = &ArrayDiInt[i];

    PuntaAMaggiore = &ArrayDiInt[0];
    for (i=1; i < SIZE; i++)
       if (ArrayDiInt[i] > *PuntaAMaggiore)
          PuntaAMaggiore = &ArrayDiInt[i];

    return 0;
}
```

Tipi e memoria occupata

- Le variabili occupano in memoria un numero di byte che dipende dal loro tipo
- Sono allocate in byte di memoria consecutivi
- L'operatore `sizeof()` restituisce il numero di byte occupati da un tipo
  - O da una variabile

```
double A[5], *p;

sizeof(double) → 8
sizeof(A[2])   → 8
sizeof(A)      → 40
sizeof(p)      → 4
sizeof(*p)     → 8
```

Aritmetica dei puntatori

- Il valore di un puntatore è sempre l'indirizzo del primo byte dello spazio occupato da una variabile
- Il C permette somme e sottrazioni tra puntatori
- Per esempio:

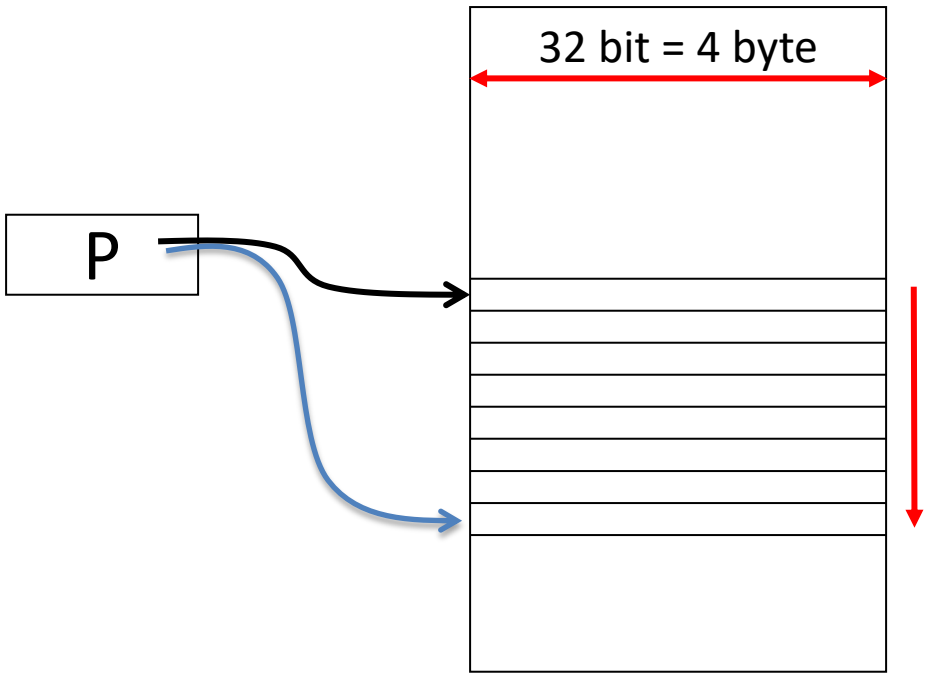
```
p += var;/* cioè  p = p + var; */
```

  - Il valore di p è incrementato di un multiplo dell'ingombro in memoria del tipo puntato, e cioè della quantità

```
var * (sizeof(TipoPuntatoDa_p))
```

Esempio

```
TipoDato *p;   / * sizeof(TipoDato) = 28 */
…
++p;
```



incremento pari a 7 parole di memoria == 28 byte

Array e puntatori

- In C esiste una parentela stretta tra array e puntatori
- Il nome di un array `v` è una costante
  - Di tipo puntatore al tipo componente l'array
  - Di valore *indirizzo della prima cella allocata per l'array*
- Esempio

  `int v[3];`
  - Definisce `v` come una costante simbolica il cui tipo è `int *const v`, cioè puntatore costante a un intero
  - Perciò `v[i]` è equivalente a `*(v + i)`
  - Calcolo dello spiazzamento nel vettore grazie all'aritmetica dei puntatori

Ecco finalmente svelato uno dei misteri della scanf!

- Perché ci vuole & per memorizzare un valore in una variabile generica, ma non in una stringa?

- La funzione scanf() riceve come parametri gli *indirizzi* delle variabili in cui scrivere i valori letti da terminale

- Gli identificatori delle variabili "normali" rappresentano la variabile, e per estrarne l'indirizzo occorre l'operatore &
  - Gli identificatori degli array rappresentano già di per sé i puntatori ai primi elementi
  - Quindi nel caso delle stringhe (che sono array) non occorre &

Riassumendo: array e puntatori

- Con la seguente dichiarazione:

```
int a[5], i, *p;
```

`a[i]`	equivale a	`*(a + i)`
`p = a;`	equivale a	`p = &a[0];`
`p = a + 1;`	equivale a	`p = &a[1];`
`a = p;`	*è un errore!*	
`a = a + 1;`	*è un errore!*	

- Occorre ricordare che a è il nome di un array
  - Cioè un *puntatore costante!*

POLITECNICO
MILANO 1863

Ancora sull'aritmetica dei puntatori

- Se p e q puntano a due diversi elementi di uno stesso array, la differenza

  `p - q` calcola la distanza nell'array tra gli elementi puntati

- Non è la differenza aritmetica tra i valori numerici dei due puntatori …

- … ma la distanza *espressa in numero di elementi*

Ancora sull'aritmetica dei puntatori

```
int x[3]={2,7,4}, *p = x;
```

Differenza tra

```
   (*p)++;
```

e

```
   *(p++);
```

Array multidimensionali

- Per calcolare lo spiazzamento occorre conoscere le dimensioni intermedie

```
Tipo m[R][C];   /* R righe, C colonne*/
```

`m[i][j]` = accesso al j-esimo elemento della i-esima riga

`m[i][j]` = `*(*(m+i)+j)` ≈ `*(m+C·i+j)`

Serve conoscere sia la dimensione del tipo sia il numero di colonne

`sizeof(Tipo)` e `C`

La "altezza" R non serve

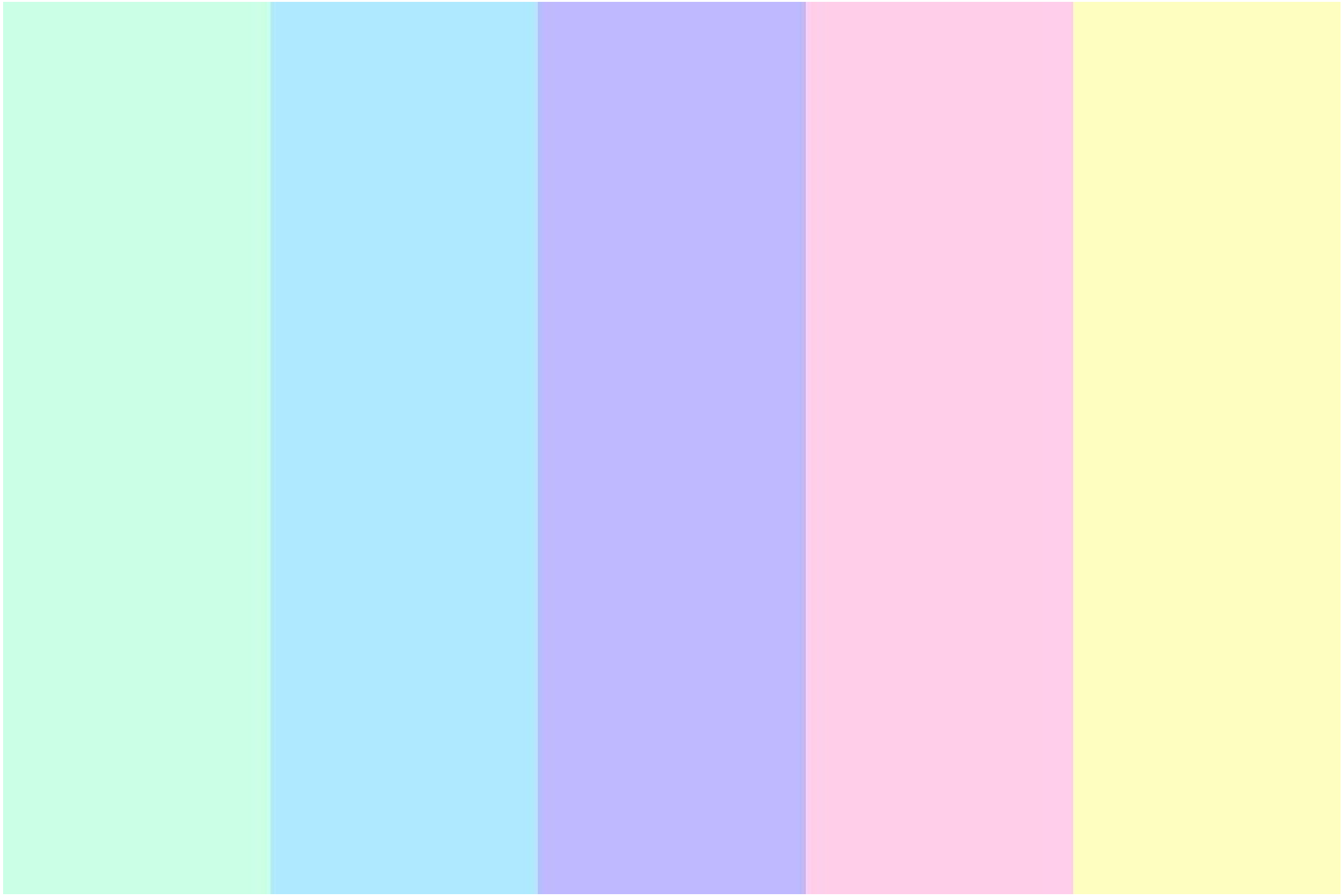
```
Tipo p[X][Y][Z]
```

`p[i][j][k]` = `*(*(*(p+i)+j)+k)` ≈ `*(p+Y·Z·i+Z·j+k)`

Serve conoscere dimensione del tipo, altezza e larghezza

`sizeof(Tipo), Y, Z`

La "profondità" X non serve



Riassumendo: array e puntatori

- Con la seguente dichiarazione:

```
int a[5], i, * p;
```

`a[i]`	equivale a `*(a + i)`
`p = a`	equivale a `p = &a[0];`
`p = a + 1`	equivale a `p = &a[1];`
`a = p;`	è un ERRORE !!
`a = a + 1;`	è un ERRORE !!

- **Cioè occorre ricordare che `a` è un array !!**

Ancora sull'aritmetica dei puntatori

- Se `p` e `q` puntano a due diversi elementi di uno stesso array, la differenza:

  `p - q`

  dà la distanza **nell'array** tra gli elementi puntati
- Tipicamente **non** coincide con la differenza "aritmetica" tra i valori numerici dei due puntatori
  - È una distanza espressa in "numero di elementi"

Array multidimensionali

Per gli array multi-dimensionali

- Il calcolo dello **spiazzamento** richiede di conoscere le dimensioni intermedie
  - **`Tipo m[R][C];`** /*N.B.: R righe, C colonne*/
  - **`m[i][j]`** → accesso al j-esimo elemento della i-esima colonna
  - **`m[i][j]`** ≡ **`*( *(m + i) + j )`** ≈ m + **C**·i + j
    - serve conoscere sia la dimensione del tipo sia il numero di colonne (sizeof(Tipo) e C; *la "altezza" R non serve*)
  - Tipo p[**X**][**Y**][**Z**]
  - p[i][j][k] ≡ *(*(*(p+i)+j)+k) ≈ p + **Y**·**Z**·i + **Z**·j + k
    - serve conoscere dimensione del tipo, altezza e larghezza (sizeof(Tipo), Y e Z; *la "profondità" X non serve*)