Se voglio puntare un intero:

Tipo: puntatore a interi

Variabile puntatore

```
int* p = NULL;
p=new int;
*p = 123;
```

Variabile intera

In questo esempio *il puntatore a interi* "**p**", non è un numero, ma *è un indirizzo*.
Il tipo di "**p**" è **int*** ed il suo indirizzo è assegnato con **p=new int**.
Il numero intero invece l'abbiamo assegnato quando facciamo ***p=123**;
Riassumendo il **numero intero *p** si trova all'**indirizzo p=new int**.
Poi quando il vettore non serve più possiamo restituire la memoria ormai inutile:

```
delete[ ]p;
```

Se voglio puntare un double si fa lo stesso procedimento:

```
double* p=NULL;
p=new double;
*p=123.45;
```

In questo esempio *il puntatore a double* "**p**", non è un numero, ma *è un indirizzo*.
Il tipo di "**p**" è **double*** ed il suo indirizzo è assegnato con **p=new double**.
Il numero con la virgola invece l'abbiamo assegnato quando facciamo ***p=123.45**;
Riassumendo il **numero decimale *p** si trova all'**indirizzo p=new double**.
Poi quando il vettore non serve più possiamo restituire la memoria ormai inutile:

```
delete[ ]p;
```

Se ho un vettore formato da 100 numeri interi invece è un po' diverso. In questo caso l'indirizzo puntato è quello del primo byte della sequenza dei 100 numeri[1]. Quindi:

```
int* p=NULL;
p=new int[100];
for(int i=0; i<100; i++)
        *(p+i)=rand();
```

È possibile semplificare il codice operando la seguente sostituzione:

```
*(p+i)≡p[i]
```

E quindi[2]:

Riserva lo spazio richiesto ed assegna l'indirizzo del primo byte del vettore

Assegna in modo casuale i 100 numeri interi del vettore

```
int* p=NULL;
p=new int[100];
for(int i=0; i<100; i++)
        p[i]=rand();
```

Poi quando il vettore non serve più possiamo restituire la memoria ormai inutile:

```
delete[ ]p;
```

[1] attenzione un numero intero occupa 4 byte, quindi dire primo byte non è equivalente a dire primo numero

[2] Dato il vettore (23, 35, 452, 12, ….) abbiamo che *p≡p[0]≡23, *(p+1)≡p[1]≡35, *(p+2)≡p[2]≡452, *(p+3)≡p[3]≡12, … e più in generale *(p+i)≡p[i].

Stesso procedimento per i vettori di stringhe:

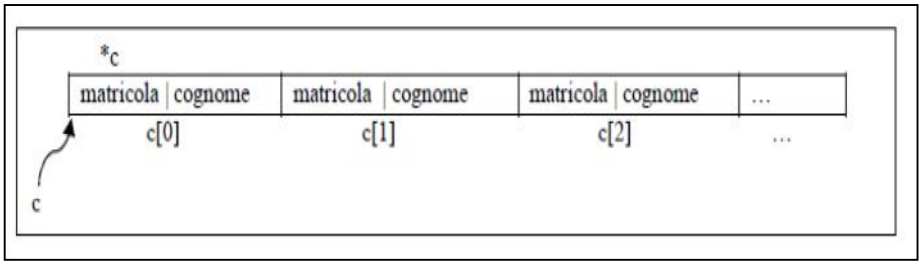
```
string* s=NULL;
s=new string[100];
for(int i=0; i<100; i++)
        cin>>s[i];
```

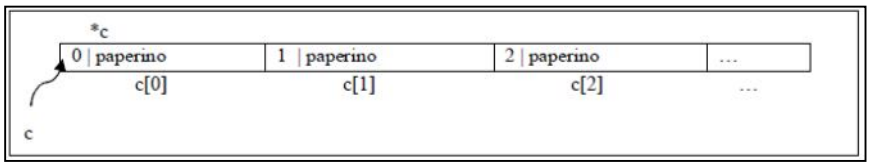
Poi quando il vettore non serve più possiamo restituire la memoria ormai inutile:

```
delete[ ]s;
```

Per le struct:

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  struct cliente
7  {
8         int matricola;
9         string cognome;
10 };
11
12 int main(int argc, char *argv[])
13 {
14     cliente* c=NULL;
15     c=new cliente[10];
16     for(int i=0; i<10; i++)
17     {
18        c[i].matricola=i;
19        c[i].cliente="paperino";
20     }
21
```





In questo esempio viene creato un vettore di struct composto da 10 clienti (e matricola) e fa puntare il suo primo elemento da **c** all'indirizzo individuato da **c=new cliente[10]**

Quando il vettore non serve più è possibile **restituire la memoria** attraverso:

```
delete[]c;
c=NULL;
```

Fino ad ora abbiamo visto come ci si comporta quando abbiamo a che fare con un solo puntatore, possiamo però aver bisogno di più puntatori e dover lavorare con vettori di puntatori, in particolare di puntatori a struct. In questo caso il codice c++ è il seguente:

```
6  struct cliente
7  {
8        int matricola;
9        string cognome;
10 };
11
12
13 int main(int argc, char *argv[])
14 {
15     cliente* vet[10]; // in questo caso ho 10 puntatori
16
17     for(int i=0; i<10; i++)
18             vet[i] = NULL;
19
20     for(int i=0; i<10; i++)
21        {
22             vet[i] = new cliente;
23             (*vet[i]).matricola = i;
24             (*vet[i]).cognome = "pluto";
25         }
```

Assegno al puntatore a struct vet[i] un indirizzo

Con davanti *, il puntatore a struct vet[i] è diventato di tipo clienti (nome della struct), cioè: il **tipo** di ***vet[i]** è **clienti**. Qui viene assegnato un numero intero alla variabile intera **matricola** che fa parte della **struct clienti**.

La sintassi del linguaggio c++ mi permette di fare la seguente sostituzione:

$$(*\ A).\ B \quad = \quad A \ \text{->}\ B$$

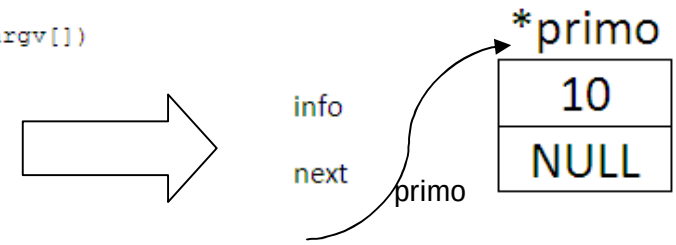
Così il codice diventa:

```
13 int main(int argc, char *argv[])
14 {
15     cliente* vet[10];
16
17     for(int i=0; i<10; i++)
18             vet[i] = NULL;
19
20     for(int i=0; i<10; i++)
21        {
22             vet[i] = new cliente;
23             vet[i] -> matricola = i;
24             vet[i] -> cognome = "pluto";
25         }
26
```

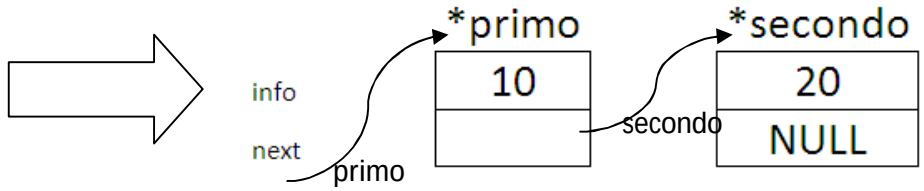
I quest'ultimo caso ho ottenuto il vantaggio di aver tenuto occupata meno RAM rispetto ai casi precedenti in cui veniva allocato un intero vettore di variabili già TUTTE disponibili.

```
1  #include <cstdlib>
2  #include <iostream>
3
4  using namespace std;
5
6  struct nodo
7  {
8   string info;
9   nodo* next;
10 };
```

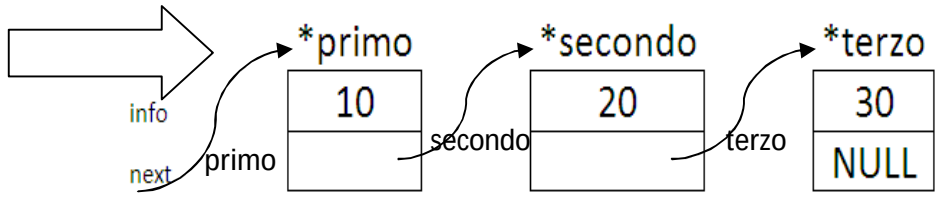
```
11
12 int main(int argc, char *argv[])
13 {
14     nodo* primo=NULL;
15     primo=new nodo;
16     primo->info=10;
17     primo->next=NULL;
18
```




```
19     nodo* secondo=NULL;
20     secondo=new nodo;
21     secondo->info=20;
22     primo->next=secondo;
23     secondo->next=NULL;
```




```
25     nodo* terzo=NULL;
26     terzo=new nodo;
27     terzo->info=30;
28     secondo->next=terzo;
29     terzo->next=NULL;
30
```



.....

Per stampare “10” (contenuto in **info** della variabile ***primo** di tipo **nodo**)

```
31
32      cout<<(*primo).info<<endl;
33
```

Che è equivalente a:

```
32      cout << primo->info << endl;
33
```

Per stampare “20”:

```
31
32      cout << (*secondo).info          << endl;
33
34      cout << (*((*primo).next)).info  << endl;
35
36      cout << (*(primo->next)).info    << endl;
37
38      cout << primo->next->info        << endl;
```

I codici scritti nei riquadri qui a sinistra sono tutti equivalenti, il migliore è l’ultimo