

**memoria  
dinamica  
e liste**

il C++ permette di **chiedere** da programma al sistema operativo **l'allocazione** di memoria da usare nel programma:

memoria per **qualsiasi** tipo, intero, double, enum, struct, array ...

**new** è la funzione che fa la richiesta, essa restituisce il puntatore alla memoria allocata:

```
int *p=new int;
```

alloca spazio Ram per un intero, p punta a questa locazione Ram

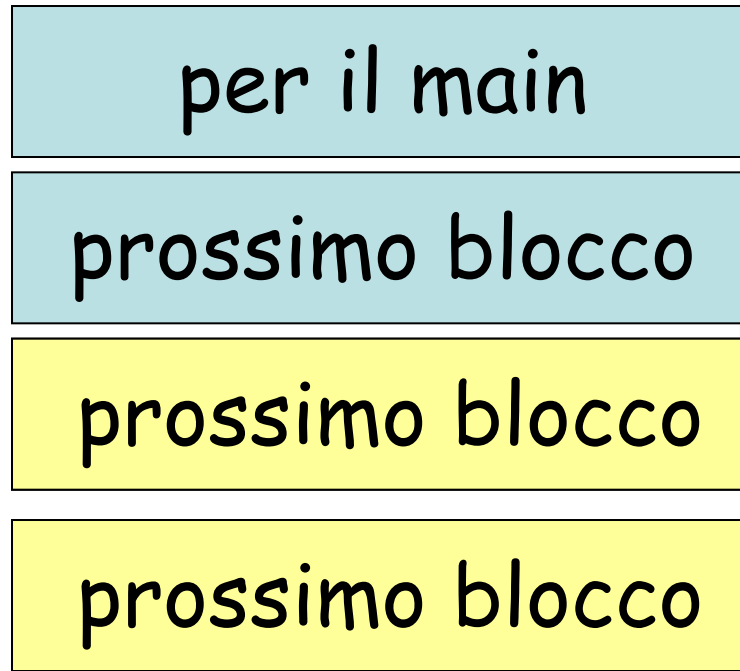
la memoria richiesta con la new viene  
allocata sullo **HEAP** che è diverso dallo  
**stack**

la deallocazione deve essere fatta  
esplicitamente da programma con la  
funzione

**delete p;**

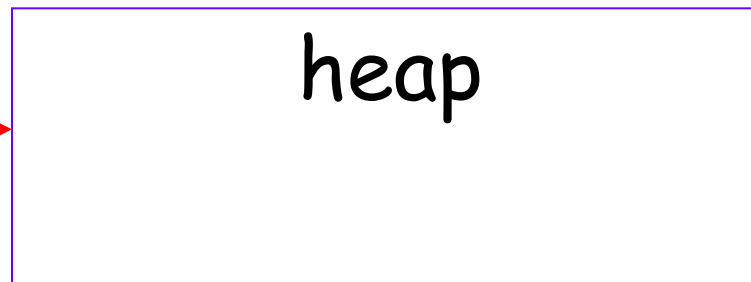
dove p punta all'oggetto da deallocare

pila dei dati  
automatici



Ram

dati  
dinamici



allocati con new e deallocati con delete

## allocazione e deallocazione di array;

```
int * p= new int[10];
```

```
delete[] p;
```

anche a più dimensioni:

```
int (*p)[10]=new int [5][10];
```

```
delete[] p;
```

```
int (*p)[8][10]=new int[5][8][10];
```

```
delete[] p;
```

possiamo adattare gli array al bisogno della particolare esecuzione del programma:

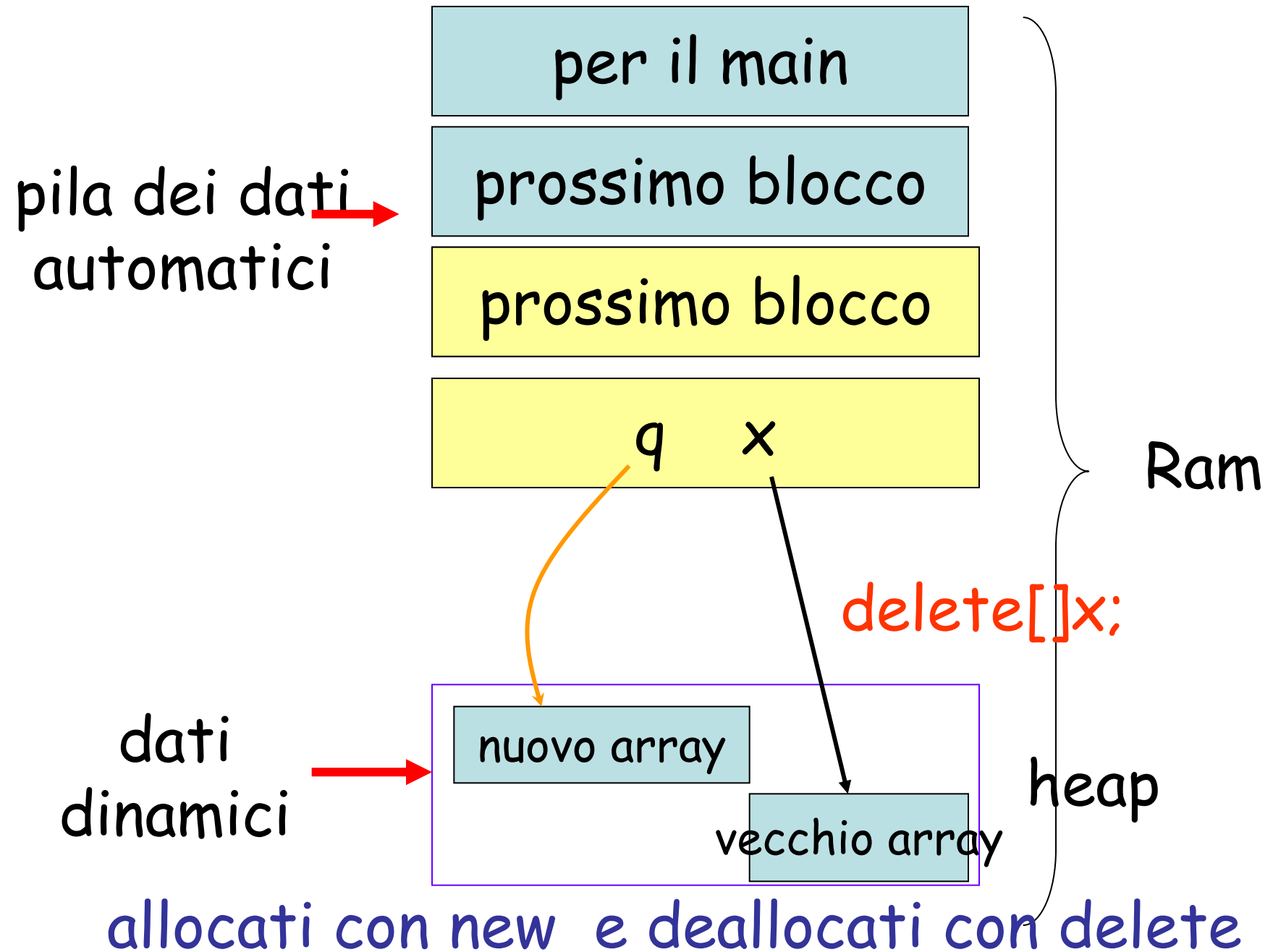
supponiamo di avere un programma che legge da cin fino alla sentinella 0 in un array che deve adattarsi a quanti interi vengono inseriti.

```
main()
{
    int *p=new int[10], dim=10, i=0;
    bool zero=false;
    while( !zero)
    {if(i==dim)
        allunga(p,dim);
        cin>>p[i];
        if(p[i]==0)
            zero=true;
        else
            i++;
    } // i valori letti,
} // p ha dim elementi
```

```
void allunga(int *& x, int & dim)
{int * q= new int[dim*2]; // nuovo
for(int i=0; i<dim; i++)
q[i]=x[i]; // ricopio il vecchio
delete [] x; // elimino vecchio
x=q; // x punta al nuovo array
dim=dim*2; // dim è nuova dimensione
}
```

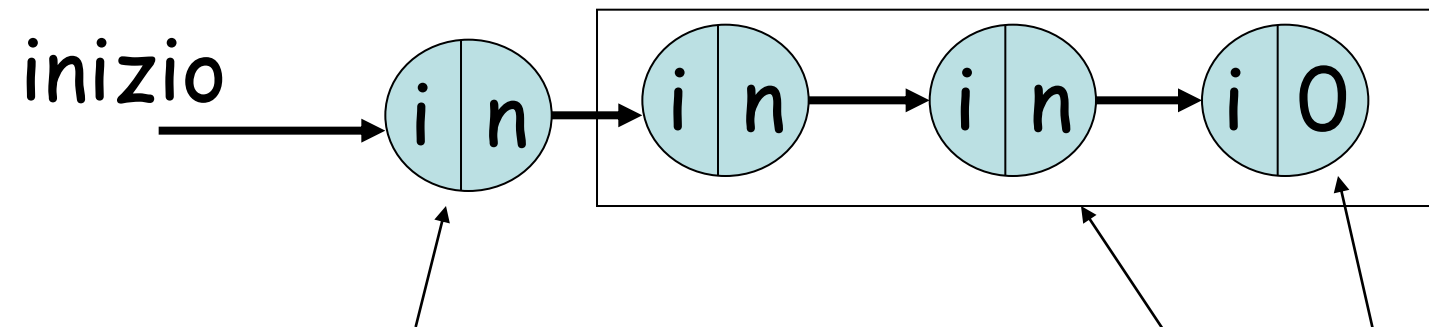
**notare:** q non va deallocata è variabile locale di **allunga** e quindi viene deallocata automaticamente (sta sulla pila)





con new e delete possiamo costruire strutture dinamiche come liste e alberi

una lista:



è un nodo che punta al resto della lista  
possiamo allungare una lista creando un nuovo  
nodo o accorciarla distruggendo un nodo  
oppure è la lista vuota  
i nodi sono allocati e presentati come punte 0  
sullo heap

pila dei dati  
automatici

per il main

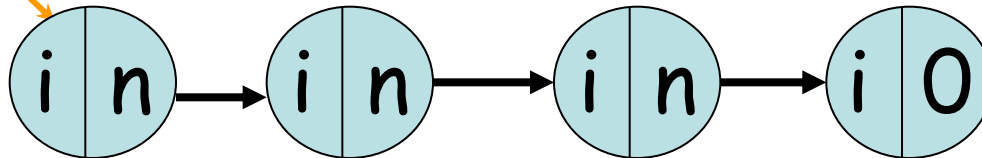
prossimo blocco

prossimo blocco

inizio

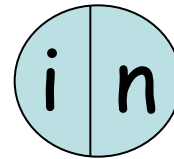
Ram

heap

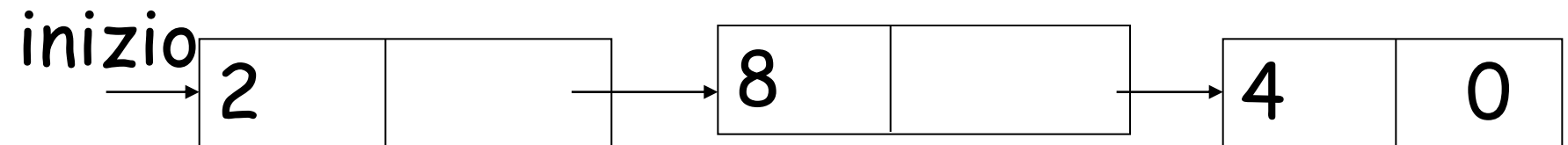


# realizzazione di una lista dinamica (lista concatenata)

ogni nodo della lista ha 2 campi:



```
struct nodo {int info; nodo *next;};
```



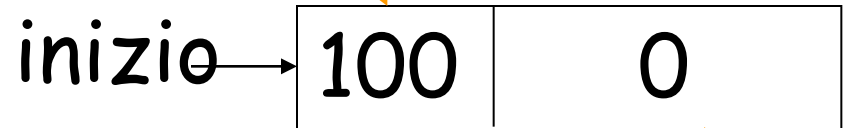
allocando i nodi dinamicamente con new ed eliminandoli con delete possiamo avere liste che crescono (si aggiungono nodi) e diminuiscono (si eliminano nodi) dinamicamente

**lista vuota:** nodo \* inizio=NULL;

inizio=new nodo;

(\*inizio).info=100;

(\*inizio).next=0;



altra notazione:

`(*inizio).info=100;`    `inizio→info=100;`

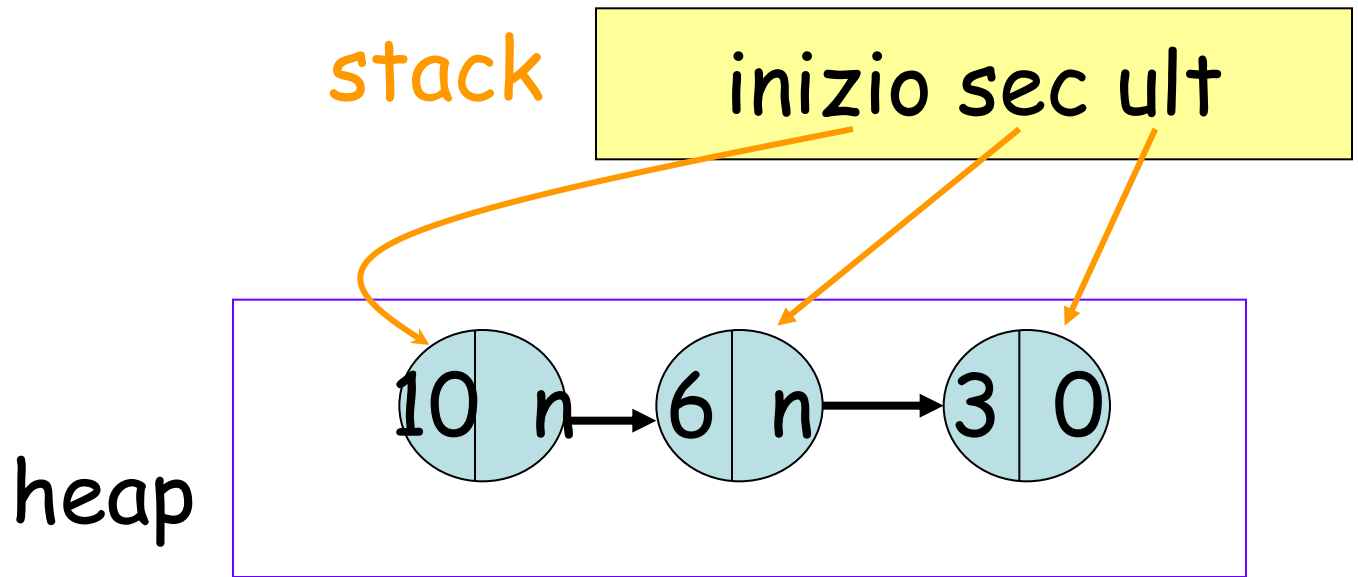
`(*inizio).next=0;`    `inizio→next=0;`

usiamo le liste dinamiche per realizzare  
la struttura dati **pila**

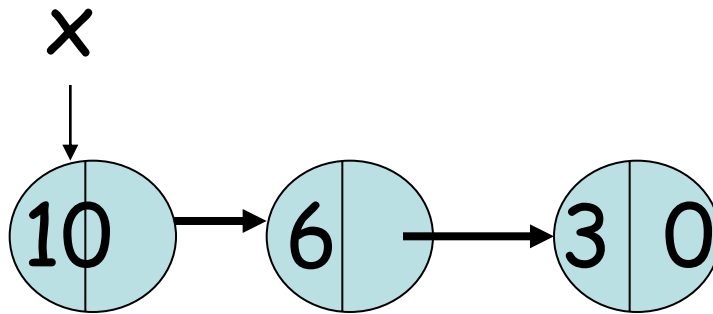
quella delle variabili automatiche dei  
programmi

```
struct nodo{int info; nodo* next;  
nodo(int a=0, nodo* b=0){info=a; next=b;}  
};
```

```
nodo * ult=new nodo(3,0);  
nodo* sec=new(6,ult);  
nodo * inizio=new nodo(10,sec);
```



```
nodo *x=inizio;  
while(x!=0)  
{  
  cout<< x->info<<endl;  
  x=x->next;  
}
```





```
void stampa(nodo *x)
{
    if(x)
    {
        cout<< x->info;
        stampa(x->next);
    }
}
```

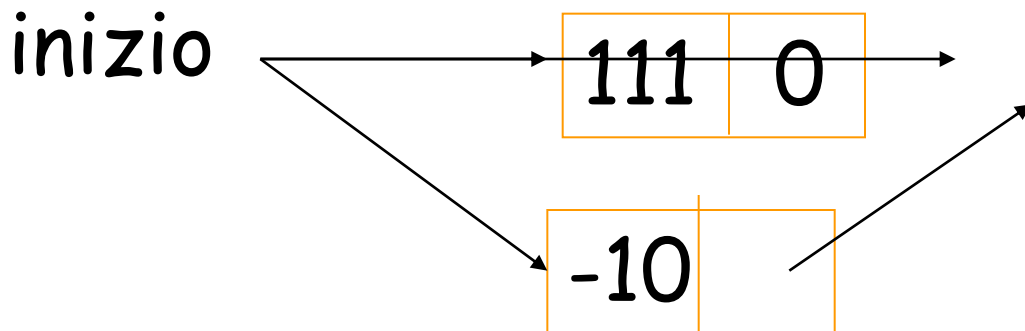
passaggio del parametro per  
valore

una pila è una pila di elementi (di tipo qualunque) su cui si possono fare 3 operazioni:

- 1) aggiungere un elemento → **push**
- 2) togliere un elemento → **pop**
- 3) testare se la pila è vuota → **empty**

pila vuota

pop



realizziamo una pila di int :

```
struct nodo{int info; nodo * next};
```

```
void push(nodo * & n, int x)
```

```
{nodo *y=new nodo;
```

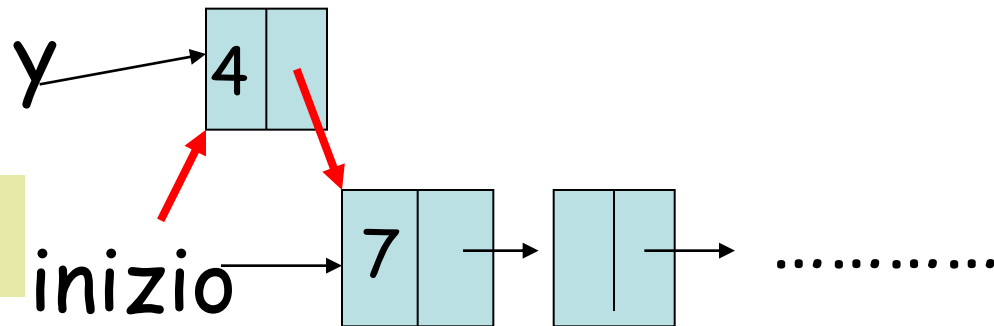
```
y→info=x;
```

```
y→next=n;
```

```
n=y;
```

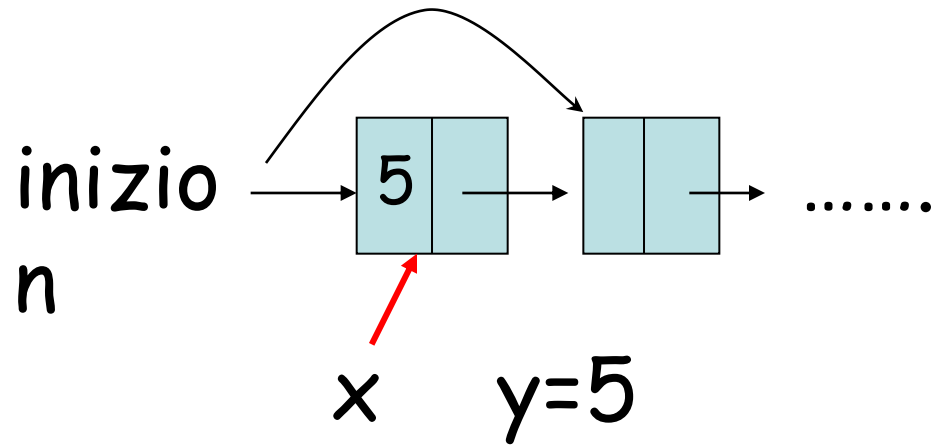
```
}
```

```
push(inizio,4);
```



```
int pop(nodo *& n)
{if(n)
{nodo *x=n;
n=n->next;
int y=x->info;
delete x;
return y;
}
else
throw(1);
}
```

`int w=pop(inizio);`



testare se una pila è vuota è semplice:

```
bool empty(nodo * n)
```

```
{
```

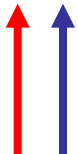
```
return n==0;
```

```
}
```

usiamo la pila per risolvere il problema  
della corretta parentesizzazione dei  
programmi

è presente nei compilatori

ogni parentesi aperta ha la  
corrispondente chiusa e viceversa:

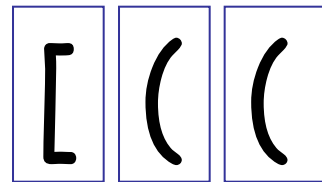
$$[(x+(y*2)*3)]*(f+6)$$


**idea:** mettiamo le parentesi aperte in una pila e quando troviamo una parentesi chiusa essa deve corrispondere con la parentesi in cima alla pila

$[(x+(y*2)*3)]$



pila



pila  
vuota  
OK



char E[..] contiene l'espressione

```
bool CP(char* E, int dim)
{
    nodo* inizio=NULL; int i=0; bool ok=true;
    for(; i < dim && ok; i++)
    {
        trattamento del carattere E[i]
    }

    if(ok && empty(inizio)) return ok;
    else return false;
}
```

trattamento di  $E[i]$

```
switch(E[i])
{
case '(': push(inizio, '('); break;
case '[': push(inizio, '['); break;
case ')': {char x=pop(inizio);
           if(x != '(' ) ok=false; break;}
case ']': {char x= pop(inizio);
           if(x != '[' ) ok=false; break;}
default: cout<< "altro carattere"<<endl;
}
```

la new può fallire !! Come l'apertura dei file

potrebbe non esserci memoria Ram sufficiente

```
int * x= new int[1000];
```

```
if(x==NULL) // la new è fallita
```

```
throw(..);
```

la costante predefinita **NULL** ha valore 0