Prossime lezioni – vedere sito del corso!

Organizzazione dei gruppi per le esercitazioni:

GRUPPO A : Matematici (matricola dispari) + Informatici Matricola divisibile per 3 + Informatici Matricola divisibile per 8
GRUPPO B: tutti gli altri informatici

Calendario delle lezioni e materiale didattico

1 - Mer 19 febbraio 8:30 aula A101 - 1 - Prima lezione
Introduzione al corso. Richiami di C. Il modello di memoria, allocazione automatica. Introduzione alla Java Virtual Machine.

- **Slides**
- **Video**

2 - Ven 21 febbraio 10:30 aula B107 - Lezione 2
Tutti gli studenti

- Lun 24 febbraio 13:30 aula B106 - Lezione *SOLO PER I MATEMATICI*

NOTA: ALLE ESERCITAZIONI 3a e 3b DOVETE ARRIVARE CON IL VOSTRO LAPTOP, SUL QUALE DOVETE AVER GIA' SCARICATO NETBEANS 8.2 (NON ALTRE VERSIONI!) - IL BUNDLE "JAVA SE" E' SUFFICIENTE. - https://netbeans.org/downloads/8.2/ NON SERVE AVERLO GIÀ ANCHE INSTALLATO.

3a - Mer 26 febbraio 8:30 aula A101 - Prima esercitazione (docente: Andrea Rosani): SOLO GRUPPO B (vedi sopra)

3b - Ven 21 febbraio 10:30 aula B107 - Prima esercitazione (docente: Andrea Rosani): SOLO GRUPPO A (vedi sopra)

Premessa: definizioni

Ne riparliamo in dettaglio (matematici)
Lunedi pomeriggio ore 13:30 aula B106

Definizione: indirizzo di memoria

indirizzo di memoria:

un identificatore univoco della posizione (locazione o cella di memoria) sulla quale il processore o un'altra periferica possono accedere per operazioni di lettura o scrittura

Tipo	nome	valore	indirizzo
int	a	1	0
int	b	2	4
int[]	c[]	7	8
		9	12
		0	16
float	d	4.0	20
			...

Operatore indirizzo &:

Dato il nome di una variabile ne restituisce l'indirizzo

z=&d; => z=20

Definizione: puntatore

Puntatore:

tipi di dato che rappresentano un indirizzo di memoria

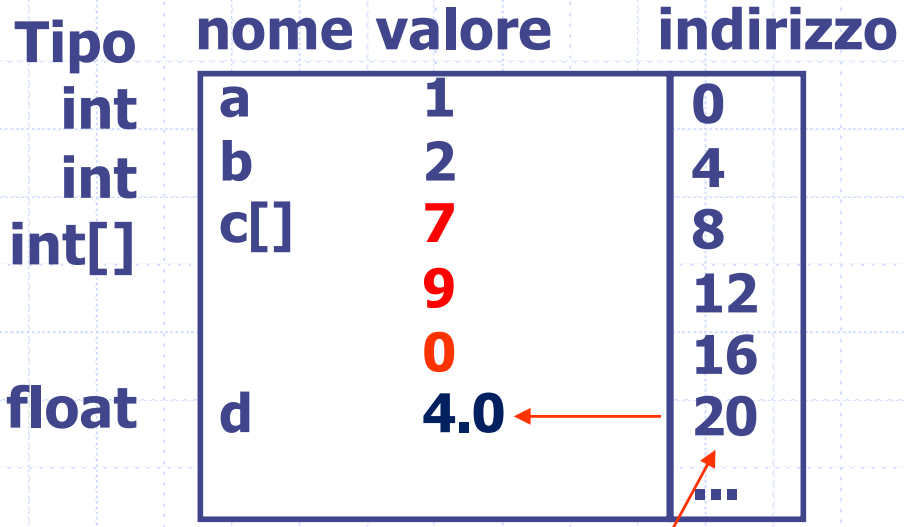
Tipo	nome	valore	indirizzo
int	a	1	0
int	b	2	4
int[]	c[]	7	8
		9	12
		0	16
float	d	4.0	20
			...

float* z=&d; => z=20

Definizione: dereferenziazione

Dereferenziazione:

Operazione per cui, dato un puntatore, si accede all'entità puntata



Definizione: struttura dati

Struttura dati (struct):

una struttura dati è un'entità usata per organizzare un insieme di dati all'interno della memoria del computer

```
struct DataTemporale {
  int giorno;
  int mese;
  int anno;
};

DataTemporale oggi;
oggi.giorno=21;
oggi.mese=2;
Oggi.anno=2020;
```

Richiami di C++ di base

- Richiami di C++ di base
- Parte 2

Funzioni: problema #1

```
void incrementa(int x) {
     x=x+1;
}
main(void) {
    int a=1;
    incrementa(a);
    cout << "a=" a << "\n";
}
```

Come faccio a scrivere una funzione che modifichi le variabili del chiamante?

Quanto vale a quando viene stampata?

I parametri sono passati per valore (copia)!

Funzioni: problema #2

Come faccio a farmi restituire
più di un valore da una funzione?

Puntatori

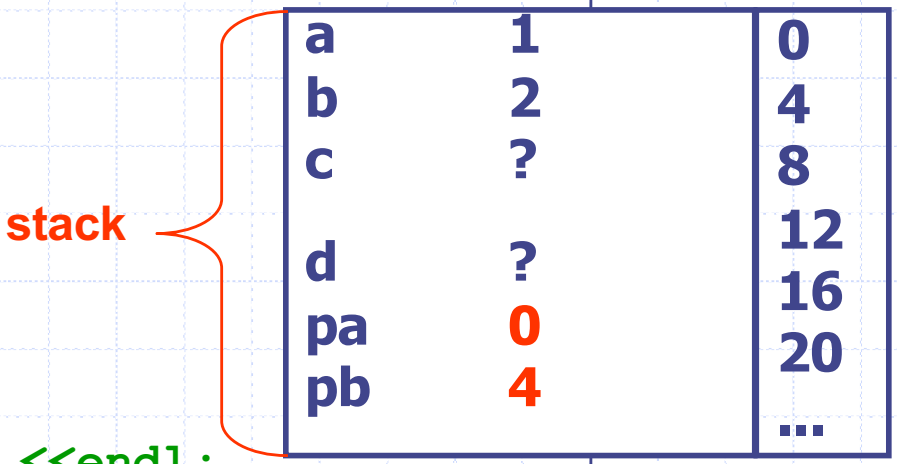
Operatore indirizzo: &

&a fornisce l'indirizzo della variabile a

Operat. di dereferenziazione: *

*p interpreta la variabile p come un puntatore (indirizzo) e fornisce il valore contenuto nella cella di memoria puntata

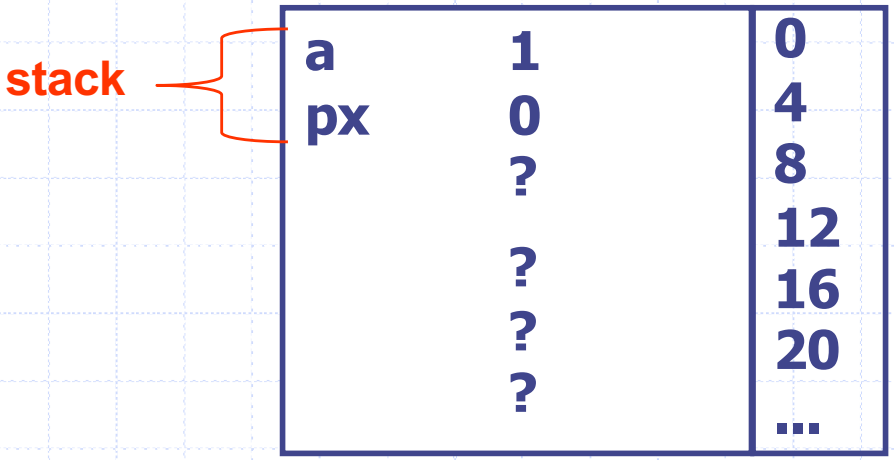
```
main() {
      int a,b,c,d;
      int * pa, * pb;
      pa=&a; pb=&b;
      a=1; b=2;
      c=a+b;
      d=*pa + *pb;
      cout << a<<"  "<<b<<"  "<< c <<endl;
      cout << a <<"  "<< *pb <<"  "<< d <<endl;
}
```



Funzioni e puntatori

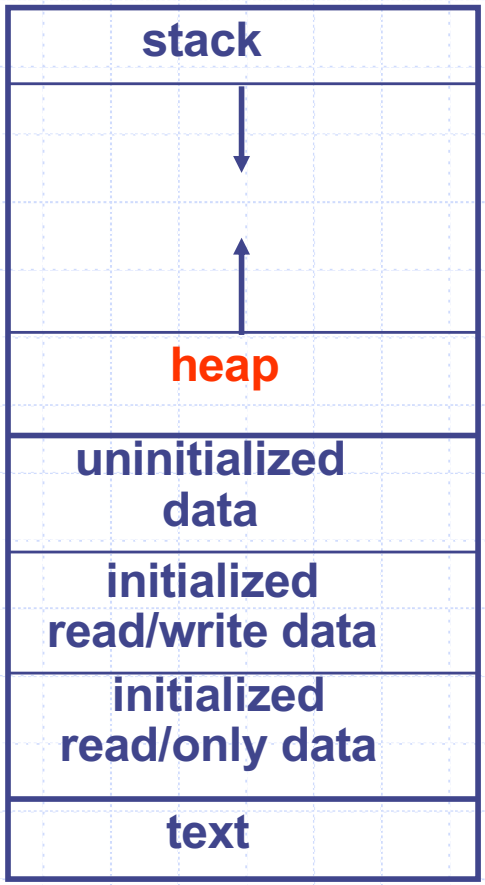
TRUCCO: per passare un parametro per indirizzo,
passiamo per valore un puntatore ad esso!

```
void incrementa(int *px) {
     *px=*px+1;
}
main(void) {
    int a=1;
    incrementa(&a);
    cout<<a<<endl;
}
```



OUTPUT: 2

Il modello di memoria



memoria allocata dalle funzioni (Variabili automatiche)

memoria allocata dinamicamente dal programmatore

Variabili globali e statiche

<- questo é supportato solo da alcuni hardware

Codice eseguibile

Operatori *new* e *delete*

`new type` alloca `sizeof(type)` bytes in memoria (heap) e restituisce un puntatore alla base della memoria allocata. (esiste una funzione simile usata in C e chiamata `malloc`)

`delete(* p)` dealloca la memoria puntata dal puntatore p. (Funziona solo con memoria dinamica allocata tramite new. Esiste un'analoga funzione in C chiamata `free`).

Il mancato uso della `delete` provoca un insidioso tipo di errore: il **memory leak**.

Allocazione della memoria

Allocazione statica di memoria (at compile time)

```
main() {
    int a;
    cout<<a<<endl; //NO!
    a=3;
    cout<<a<<endl;
}
```

```
OUTPUT:    1
           3
```

Allocazione dinamica di memoria (at run time)

```
main() {
    int *pa;
    pa=new int;
    cout<<*pa<<endl; //NO!
    *pa=3;
    cout<<*pa<<endl;
    delete(pa);
    cout<<*pa<<endl; //NO!
}
```

```
OUTPUT:    4322472
           3
           8126664
```

Vettori rivistati

Dichiarare un vettore è in un certo senso come dichiarare un puntatore.

```
v[0]    è equivalente a      *v
```

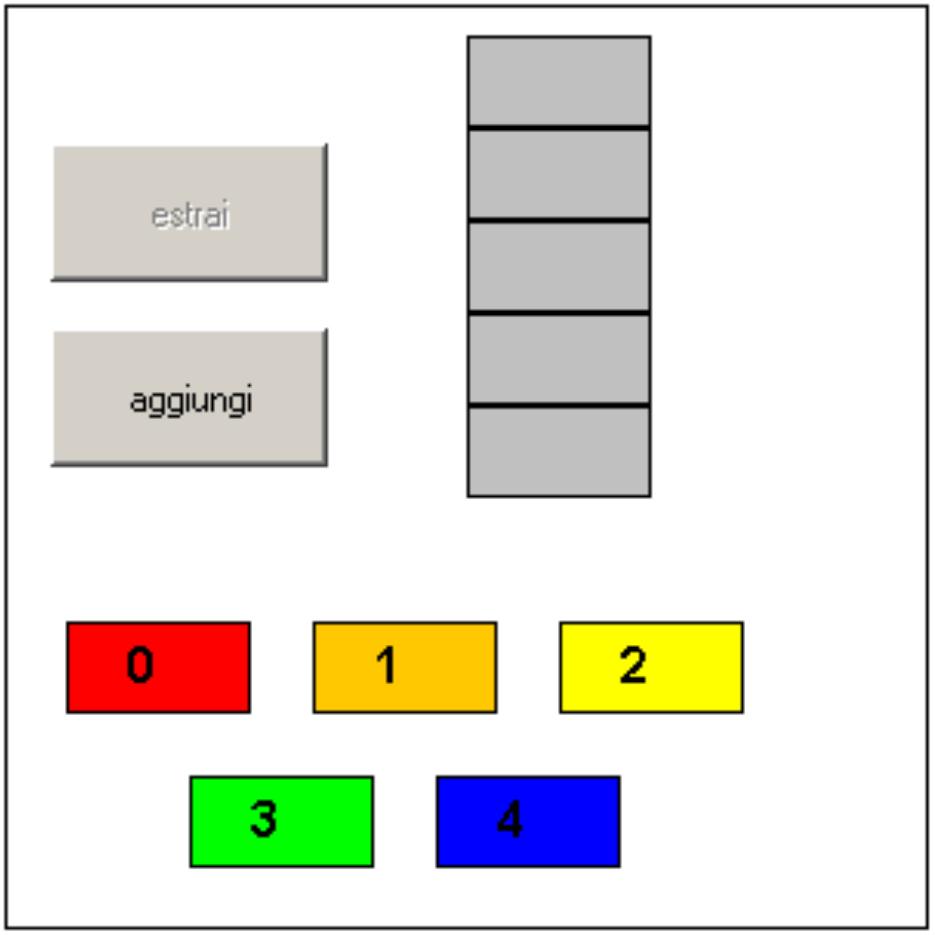
Attenzione però alla differenza!

```
int v[100];     è "equivalente" a:
                                  int *v; v=new int[100];
```

ATTENZIONE!
la prima versione alloca spazio STATICAMENTE (Stack)
la seconda versione alloca spazio DINAMICAMENTE (Heap)

Dal C alla OOP (con C++ e Java) attraverso un esempio

Costruiamo uno stack (pila)



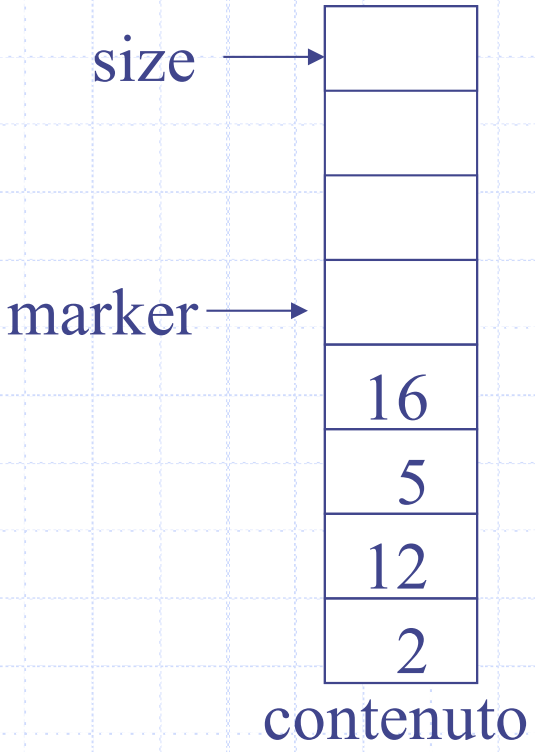
stackapplet.html

Definizione del tipo `Pila`

```
#include <iostream.h>
#include <cassert>

const int growthSize=5;

struct Pila {
   int size;
   int marker;
   int* contenuto;
};
```



Creare (e inizializzare!) una nuova pila

```
Pila* crea(int initialSize) {
//crea una Pila
  cout << "entro in crea" << endl;
  Pila *s = new Pila ;
  s->size = initialSize;
  s->marker = 0;
  s->contenuto = new int[initialSize];
 return s;
}
```

Ricorda: s->size significa (*s).size

Distruggere una pila esistente

```
void distruggi(Pila *s) {
cout << "entro in distruggi" << endl;
  delete[](s->contenuto);
  delete s;
}
```

Espandere una pila

```
void cresci(Pila *s, int increment){
   cout << "entro in cresci" << endl;
   s->size += increment;
   int* temp = new int[s->size];
   for(int k=0; k < s->marker; k++) {
      temp[k] = s->contenuto[k];
   }
   delete[](s->contenuto);
   s->contenuto = temp;
}
```

Inserire un valore in cima alla pila (*push*)

```
void inserisci(Pila *s, int k) {
   cout << "entro in inserisci" << endl;
   if(s->size == s->marker)
       cresci(s, growthSize);
   s->contenuto[s->marker] = k;
   s->marker++;
}
```

Estrarre il valore in cima alla pila (*pop*)

```
int estrai(Pila *s) {
   cout << "entro in estrai" << endl;
   assert(s->marker>0);
   return s->contenuto[--(s->marker)];
}
```

`assert`: permette di verificare che una proprietà del programma necessaria a proseguire l'esecuzione (***precondizione*** o ***asserzione***) sia vera:
se non lo è, l'esecuzione termina.

Stampare informazioni sulla pila

```
void stampaStato(Pila *s) {
   cout <<"====================="<< endl;
   cout << "size = " << s->size << endl;
   cout << "marker = " << s->marker << endl;
   for(int k=0; k < s->marker; k++)
      cout << "[" << (s->contenuto[k]) << "]";
   cout << endl;
   cout << "====================" << endl;
}
```

Creare una copia di una pila esistente

```
Pila* copia(Pila *from) {
   cout << "entro in copia" << endl;
   Pila *to = crea(from->size);
   for(int k=0; k < from->marker; k++)
      to->contenuto[k] = from->contenuto[k];
   to->marker = from->marker;
   return to;
}
```

Un programma di test

```
int main() {
  Pila *s = crea(5);
  cout << "s"; stampaStato(s);
  for(int k=1; k<10; k++)
    inserisci(s, k);
  cout << "s"; stampaStato(s);
  Pila *w = copia(s);
  cout << "w"; stampaStato(w);
  for (int k=1; k<8;k++)
    cout << estrai(s) << endl;
  cout << "s"; stampaStato(s);
  distruggi(s);
  cout << "s"; stampaStato(s);
  for(int k=1; k<15; k++)
     cout << estrai(w) << endl;
  cout << "w"; stampaStato(w);
}
```

```
bash-2.02$ g++  Pila.cpp  -o Pila.exe
bash-2.02$ Pila.exe
entro in crea
s====================
size = 5
marker = 0

====================
entro in inserisci
entro in inserisci
entro in inserisci
entro in inserisci
entro in inserisci
entro in inserisci
entro in cresci
entro in inserisci
entro in inserisci
entro in inserisci
```

```
s====================
size = 10
marker = 9
[1][2][3][4][5][6][7][8][9]
====================
entro in copia
entro in crea
w====================
size = 10
marker = 9
[1][2][3][4][5][6][7][8][9]
====================
```

```
entro in estrai
9
entro in estrai
8
…
entro in estrai
4
entro in estrai
3
s====================
size = 10
marker = 2
[1][2]
====================
```

```
entro in distruggi
s====================
size = 1627775824
marker = 2
[1627775848][1627775848]
====================
entro in estrai
9
entro in estrai
8
…
entro in estrai
2
entro in estrai
1
entro in estrai
Assertion failed: (s->marker>0),
function estrai, file Pila.cpp, line 56.
bash-2.02$
```

Ma perchè abbiamo scritto il metodo **copia**?

```
#include <Pila.h>
int main() {
   Pila  * s=crea(5);
    cout<<"s"; stampaStato(s);

    for (int k=1; k<10;k++) inserisci(s,k);
    cout<<"s"; stampaStato(s);
    Pila * w=s;
    cout<<"w"; stampaStato(w);
    for (int k=1; k<8;k++)
      cout<< estrai(s)<<endl;
    cout<<"s"; stampaStato(s);
    cout<<"w"; stampaStato(w);
}
```

Una copia (troppo) sbrigativa ...

s==========

size = 10

marker = 9

[1][2] [3][4] [5] [6] [7] [8] [9]

==========

w==========

size = 10

marker = 9

[1][2] [3][4] [5] [6] [7] [8] [9]

==========

entro in estrai

9

entro in estrai

8

…

…

entro in estrai

4

entro in estrai

3

s==========

size = 10

marker = 2

[1][2]

==========

w==========

size = 10

marker = 2

[1][2]

==========

Interfaccia vs. implementazione: Pila.h

```
struct Pila {
   int size;
   int marker;
   int *contenuto;
};

Pila* crea(int initialSize);
void distruggi(Pila *s);
Pila* copia(Pila *from);
void cresci(Pila *s, int increment);
void inserisci(Pila *s, int k);
int estrai(Pila *s);
void stampaStato(Pila *s);
```

Descrive solo la struttura dati e le funzioni per manipolarla, ma non la loro implementazione

Problemi

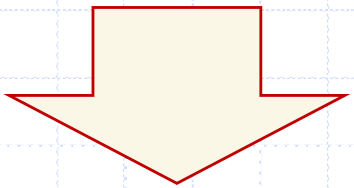
- In questo modo però la struttura dati è slegata dalle funzioni che la manipolano
- Per programmi di grandi dimensioni, diventa difficile capire quali funzioni sono "parte integrante" di una struttura dati, e quali invece semplicemente la usano (es., passaggio parametri)
- In fondo, una **`struct`** non è altro che una "collezione" di variabili legate fra loro da un ***tipo*** che le contiene. Aggiungiamo ad esse anche le (sole) funzioni che le manipolano.

Pila.h – verso una nuova versione

```
struct Pila {
   int size;
   int marker;
   int *contenuto;
   int estrai();
};
Pila* crea(int initialSize);
void distruggi(Pila *s);
Pila* copia(Pila *from);
void cresci(Pila *s, int increment);
void inserisci(Pila *s, int k);
// int estrai(Pila *s); vecchia versione
void stampaStato(Pila *s);
```

Re-implementazione di **estrai**

```
int estrai(Pila *s) {
   cout << "entro in estrai" << endl;
   assert(s->marker>0);
   return s->contenuto[--(s->marker)];
}
```



```
int estrai() {
   cout << "entro in estrai" << endl;
   assert(this->marker>0);
   return this->contenuto[--(this->marker)];
}
```

Re-implementazione di **main**

```
int main() {
   Pila *s = crea(5);
    cout << "s"; stampaStato(s);
    for (int k=1; k<10; k++)
      inserisci(s,k);
    cout << "s"; stampaStato(s);
    Pila *w = copia(s);
    cout << "w"; stampaStato(w);
    for(int k=1; k<8; k++)
      //cout << estrai(s) << endl;
      cout << s->estrai() << endl;
    …
}
```

Dove scrivere il codice di **estrai**?

Alternativa #1

```
struct Pila {
   int size;
   int marker;
   int *contenuto;
   int estrai() {
      //estrai l'ultimo valore
      cout << "entro in estrai" << endl;
      assert(this->marker>0);
      return this->contenuto[--(this->marker)];
   }
};
```

Dove scrivere il codice di **estrai**?

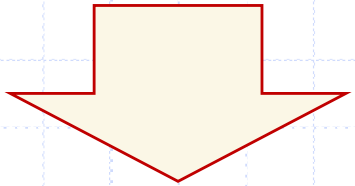
Alternativa #2

```
struct Pila {
   int size;
   int defaultGrowthSize;
   int marker;
   int *contenuto;
   int estrai();
};

int Pila::estrai() {
   //estrai l'ultimo valore
   cout << "entro in estrai" << endl;
   assert(this->marker>0);
   return this->contenuto[--(this->marker)];
}
```

`this` può rimanere implicito...

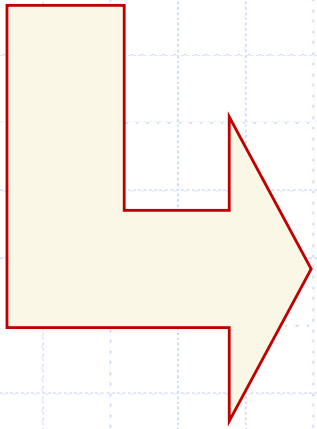
```
int estrai(Pila *s) {
//estrai l'ultimo valore
  cout<<"entro in estrai"<<endl;
  assert(s->marker>0);
  return s->contenuto[--(s->marker)];
}
```



```
int estrai() {
//estrai l'ultimo valore
  cout<<"entro in estrai"<<endl;
  assert(marker>0);
  return contenuto[--(marker)];
}
```

Re-implementazione di **crea**

```
Pila* crea(int initialSize) {
  Pila *s = new Pila ;
  s->size=initialSize;
  s->marker=0;
  s-> contenuto=new int[initialSize];
 return s;
}
```



```
Pila::Pila(int initialSize) {
  size = initialSize;
  marker=0;
  contenuto = new int[initialSize];
}
```

costruttore

Re-implementazione di **distruggi**

```
void Pila::distruggi() {
  //distruggi la Pila
  cout << "entro in distruggi" << endl;
  delete []contenuto;
  delete this;
}
```

```
Pila::~Pila() {
  //distruggi la Pila
  cout << "entro nel distruttore" << endl;
  delete []contenuto;
  // delete this;  -- NO!!
}
```

distruttore

Re-implementazione di **main**

```
int main() {
   Pila *s = new Pila(5); // OLD: = crea(5)
    cout << "s"; s->stampaStato();
    for (int k=1; k<10; k++) s->inserisci(k);
    cout << "s"; s->stampaStato();
    Pila *w = s->copia();
    cout << "w"; w->stampaStato();
    for (int k=1; k<8; k++)
       cout << s->estrai()<< endl;
    cout << "s"; s->stampaStato();
    delete s; // OLD: s->distruggi();
    cout << "s"; s->stampaStato();
    for (int k=1; k<15; k++)
        cout << w->estrai() << endl;
    cout << "w"; w->stampaStato();
}
```

`Pila.h` – una nuova versione

```
struct Pila {
   int size;
   int marker;
   int *contenuto;
   Pila(int initialSize);
   ~Pila();
   Pila* copia();
   void cresci(int increment);
   void inserisci(int k);
   int estrai();
   void stampaStato();
};
```

variabili di istanza
(o dati membro)

metodi,
(o funzioni membro)

Problemi

- Ora la struttura dati è associata in maniera chiara alle funzioni che la manipolano
- Tuttavia, nulla vieta al programmatore di accedere direttamente alla struttura dati interna di una variabile di tipo **`Pila`**
  - Ad esempio, nel main potrei scrivere:
    ```
    Pila *s = new Pila();
    s->marker = 15;
    ```
  - Questo è pericoloso: consente all'utente di **`Pila`** di aggirare le funzioni fornite dal suo autore
- Viola i principi di Parnas, lasciando accesso all'utente di **`Pila`** più di quanto necessario

Incapsulamento & *information hiding*

```
struct Pila {
    Pila(int initialSize);
    Pila();
    ~Pila();
    Pila* copia();
    void inserisci(int k);
    int estrai();
    void stampaStato();
  private:
    int size;
    int marker;
    int *contenuto;
    void cresci(int increment);
};
```

Quanto segue è accessibile solo dall'interno della variabile di tipo **`Pila`**, ma non dall'esterno

`struct` oppure `class`?

```
class Pila {
    int size;
    int defaultGrowthSize;
    int marker;
    int *contenuto;
    void cresci(int increment);
  public:
    Pila(int initialSize) ;
    Pila();
    ~Pila();
    Pila* copia();
    void inserisci(int k);
    int estrai();
    void stampaStato();
};
```

Per default, dati/funzioni sono **`private`**

... ma possono essere rese disponibili a tutti

`struct` oppure `class`?

```
struct Pila {
 private:
  int size;
  int marker;
  int *contenuto;
  void cresci(int increment);
 public:
  Pila(int initialSize);
  Pila();
  ~Pila();
  Pila* copia();
  void inserisci(int k);
  int estrai();
  void stampaStato();
};
```

```
class Pila {
 private:
  int size;
  int marker;
  int *contenuto;
  void cresci(int increment);
 public:
  Pila(int initialSize);
  Pila();
  ~Pila();
  Pila* copia();
  void inserisci(int k);
  int estrai();
  void stampaStato();
};
```

La differenza principale è la visibilità di *default* di dati/funzioni membro: per variabili **`struct`** è **`public`**, per variabili **`class`** è **`private`**