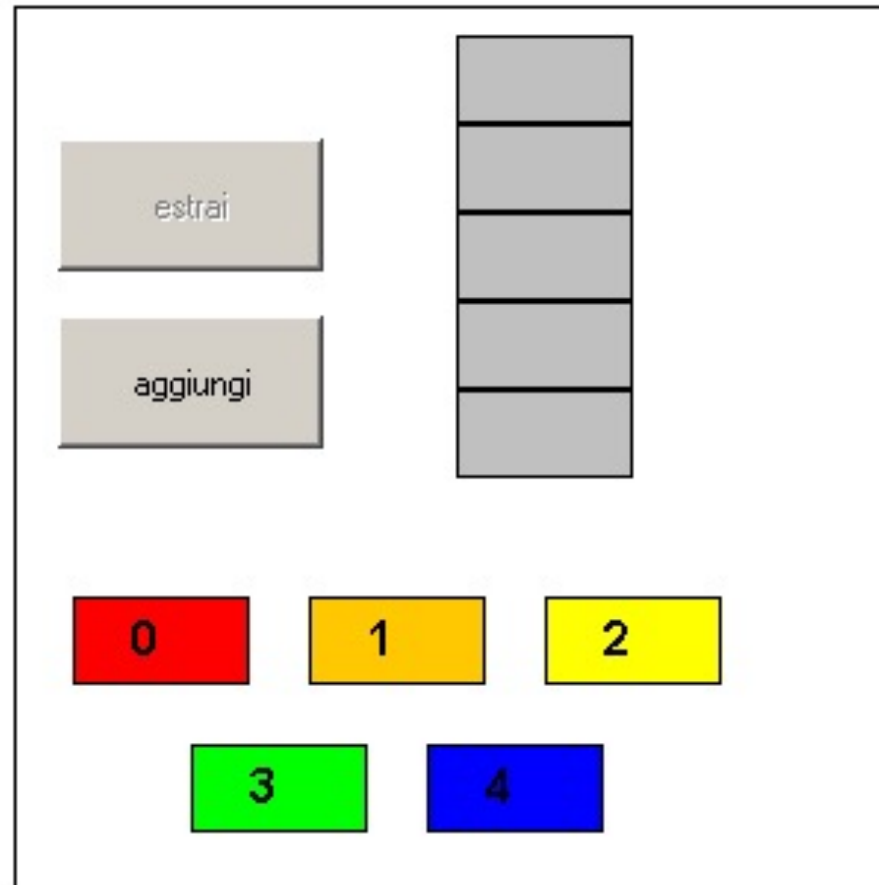


Dal C alla OOP (con C++ e Java) attraverso un esempio

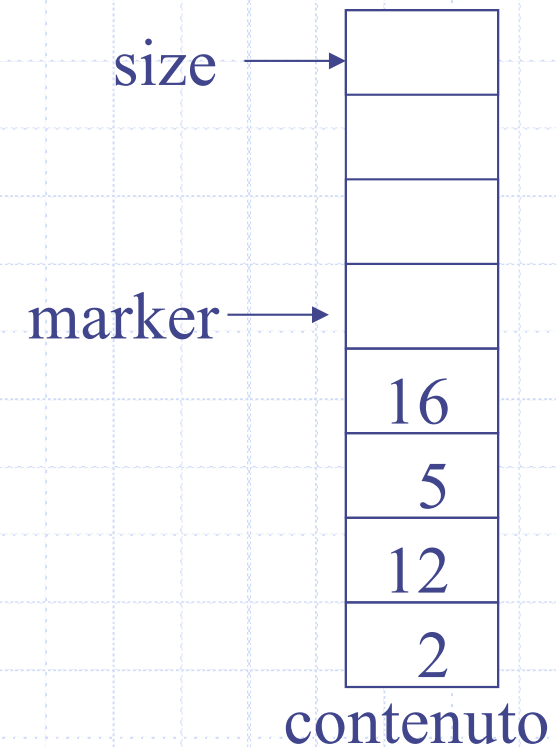
**Costruiamo uno
stack (pila)**



Definizione del tipo Pila

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

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



3

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) {  
    const int growthSize=5;  
    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

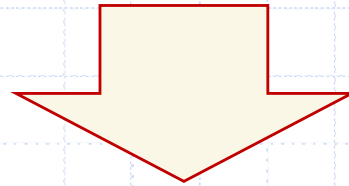
- ◆ In questo modo però la struttura dati è slegata dalle funzioni che la manipolano
- ◆ Per programmi di grandi dimensioni, diventa difficile capire quali funzioni sono “parte integrante” di una struttura dati, e quali invece semplicemente la usano (es., passaggio parametri)
- ◆ In fondo, una **struct** 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`?

```
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)];  
    }  
};
```

Alternativa #1

Dove scrivere il codice di `estrai`?

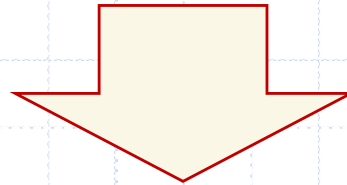
```
struct Pila {  
    int size;  
    int marker;  
    int *contenuto;  
    int estrai();  
};
```

Alternativa #2

```
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...

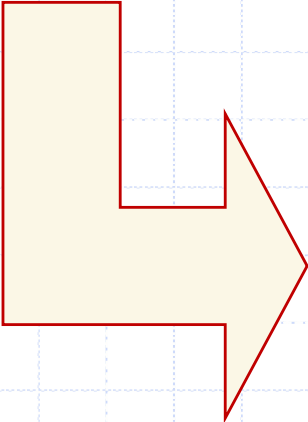
```
int estrai(Pila *s) {  
    //estrai l'ultimo valore  
    cout<<"entro in estrai"<<endl;  
    assert(s->marker>0);  
    return s->contenuto[--(s->marker)];  
}
```



```
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;  
}
```

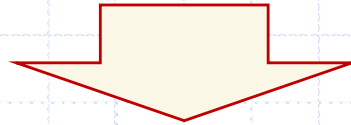


```
Pila::Pila(int initialSize) {  
    size = initialSize;  
    marker=0;  
    contenuto = new int[initialSize];  
}
```

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**