

Laboratorio di Programmazione
Corso di Laurea in Informatica
Gr. 3 (N-Z)
Università degli Studi di Napoli Federico II

A.A. 2022/23
A. Apicella

Politica di accesso

Politica di accesso: insieme di regole utilizzate per accedere ad una struttura dati (e.g., array o liste) in lettura/scrittura.

Nella loro versione base, le strutture dati forniscono *accesso libero*: l'utilizzatore può accedere liberamente a *qualsiasi* porzione della struttura dati senza alcuna regola che ne limiti l'accesso.

Esempio: in un array, è possibile accedere a qualsiasi posizione in lettura/scrittura attraverso l'operatore []

Politica di accesso

Esempio

```
int main()
{
    int V[10];
    V[3] = 5;
    V[1] = 2;
    V[7] = 44;

    return 0;
}
```

nessun problema ad accedere a qualsiasi posizione dell'array nel corso del programma. Posso accedere alla prima così come posso accedere all'ultima così come posso accedere alla posizione centrale e così via, in maniera libera.

Politica di accesso

Esempio

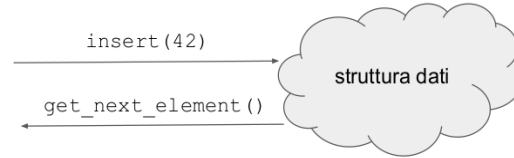
```
int main()
{
    int V[10];
    V[3] = 5;
    V[1] = 2;
    V[7] = 44;

    return 0;
}
```

nessun problema ad accedere a qualsiasi posizione dell'array nel corso del programma. Posso accedere alla prima così come posso accedere all'ultima così come posso accedere alla posizione centrale e così via, in maniera libera.

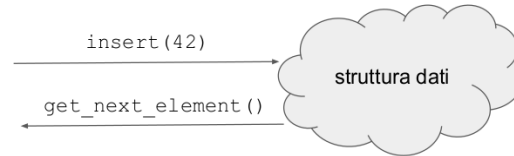
Tuttavia, strutture dati con politiche di accesso "libere" come gli array non sempre sono la scelta migliore. In generale, esistono strutture dati con politiche di accesso più ristrette, ossia che seguono determinate regole.

Politica di accesso



In generale possono esistere strutture dati che forniscono funzioni per accedervi il lettura/scrittura, ma senza dare all'utente la possibilità di specificare dove e come inserire/leggere i valori

Politica di accesso



In generale possono esistere strutture dati che forniscono funzioni per accedervi il lettura/scrittura, ma senza dare all'utente la possibilità di specificare dove e come inserire/leggere i valori

Esempi:

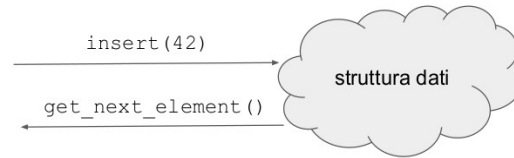
- Politica di accesso **Coda (Queue)**:

- è possibile accedere in lettura soltanto all'elemento meno recente inserito nella struttura dati. L'eventuale rimozione del primo elemento permetterà di leggere (ed eventualmente rimuovere) il successivo, e così via.
- detta anche modalità di accesso **FIFO (First In, First Out)**

- Politica di accesso **Pila (Stack)**:

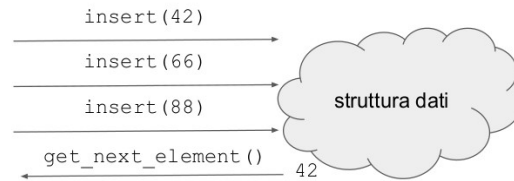
- è possibile accedere in lettura soltanto all'elemento più recente inserito nella struttura dati. L'eventuale rimozione del primo elemento permetterà di leggere (ed eventualmente rimuovere) il successivo, e così via.
- detta anche modalità di accesso **LIFO (Last In, First Out)**

Politica di accesso



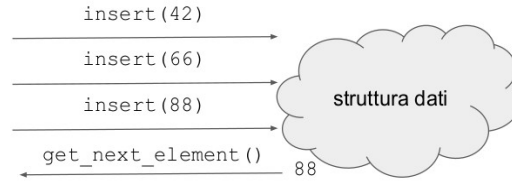
Politica di accesso

Esempio: coda

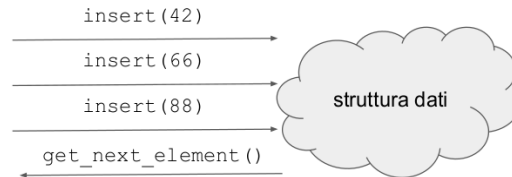


Politica di accesso

Esempio: pila



Politica di accesso



Il C non mette a disposizione in maniera nativa strutture dati che implementano simili politiche di accesso. E' però possibile *costruirle*, utilizzando ciò che si ha a disposizione (e dandoci noi qualche regola).

Una struttura dati che adotta una particolare politica di accesso deve quindi essere composta da:

- una struttura dati d'appoggio che conterrà materialmente i dati (esempio: un array o una lista)
- un insieme di **funzioni** ad-hoc per accedere alla struttura dati rispettando le regole della politica di accesso

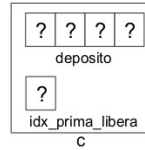
L'accesso a tale struttura deve quindi essere effettuato **unicamente** tramite le funzioni definite.

Accedervi in qualsiasi altro modo è una violazione della politica d'accesso!

Politica di accesso

Possibile implementazione di una coda

```
#define N_CODA 4
struct Coda
{
    int deposito[N_CODA];
    int idx_prima_libera;
};
```



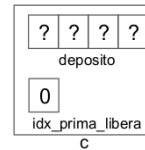
```
int main()
{
    struct Coda c;
}
```

Politica di accesso

Possibile implementazione di una coda

```
#define N_CODA 4
struct Coda
{
    int deposito[N_CODA];
    int idx_prima_libera;
};

void init_queue (struct Coda* c)
{
    c->idx_prima_libera = 0;
}
```

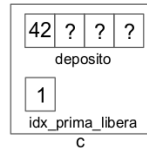


```
int main()
{
    struct Coda c;
    init_queue(&c);
}
```

Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int insert_in_queue(struct Coda* c,
                    int elemento)
{
    if (c->idx_prima_libera >= N_CODA)
    {
        return ERR;
    }
    c->deposito[c->idx_prima_libera] = elemento;
    c->idx_prima_libera++;
    return OK;
}
```



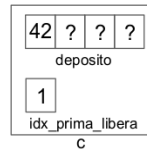
0
err

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
}
```

Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int insert_in_queue(struct Coda* c,
                    int elemento)
{
    if (c->idx_prima_libera >= N_CODA)
    {
        return ERR;
    }
    c->deposito[c->idx_prima_libera] = elemento;
    c->idx_prima_libera++;
    return OK;
}
```



0
err

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1); //NB: da gestire bene!!!
}
```

Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int read_from_queue(struct Coda* c,
                    int* p_val)
{
    if(c->idx_prima_libera == 0)
    {
        return ERR;
    }
    *p_val = c->deposito[0];
    return OK;
}
```

→

42	66	?	?
deposito			
2	idx_prima_libera		
c			

0
err
42
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = read_from_queue(&c, &val);
}
```

Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                  int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }

    c->idx_prima_libera--; //ho una posizione libera in più
    return OK;
}
```

→

42	66	?	?
deposito			
2	idx_prima_libera		
c			

0
err
42
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
}
```

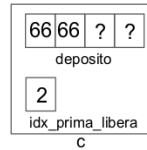

Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                   int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }
    c->idx_prima_libera--; //ho una posizione libera in più
    return OK;
}
```



0
err
42
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
}
```

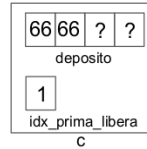
Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                   int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }
    c->idx_prima_libera--; //ho una posizione libera in più
    return OK;
}
```



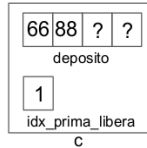
0
err
42
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
}
```

Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int insert_in_queue(struct Coda* c,
                    int elemento)
{
    if (c->idx_prima_libera >= N_CODA)
    {
        return ERR;
    }
    c->deposito[c->idx_prima_libera] = elemento;
    c->idx_prima_libera++;
    return OK;
}
```



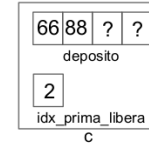
0
err
42
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
}
```

Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int insert_in_queue(struct Coda* c,
                    int elemento)
{
    if (c->idx_prima_libera >= N_CODA)
    {
        return ERR;
    }
    c->deposito[c->idx_prima_libera] = elemento;
    c->idx_prima_libera++;
    return OK;
}
```



0
err
42
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
}
```

Politica di accesso

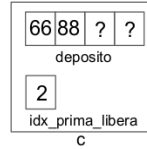
Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                  int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }

    c->idx_prima_libera--; //ho una posizione libera in più
    return OK;
}
```



0
err
66
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
    err = get_from_queue(&c, &val);
}
```

Politica di accesso

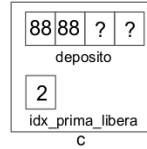
Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                  int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }

    c->idx_prima_libera--; //ho una posizione libera in più
    return OK;
}
```



0
err
66
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
    err = get_from_queue(&c, &val);
}
```

Politica di accesso

Possibile implementazione di una coda

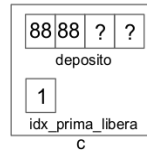
```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                  int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }
}
```

→ c->idx_prima_libera--; //ho una posizione libera in più
return OK;

}



0
err
66
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
    err = get_from_queue(&c, &val);
}
```

Politica di accesso

Possibile implementazione di una coda

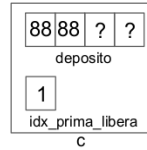
```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                  int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }
}
```

→ c->idx_prima_libera--; //ho una posizione libera in più
return OK;

}



0
err
88
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
    err = get_from_queue(&c, &val);
    err = get_from_queue(&c, &val);
}
```

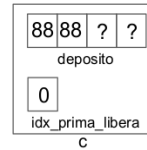
Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                  int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }
}
```



0
err
88
val

```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
    err = get_from_queue(&c, &val);
    err = get_from_queue(&c, &val);
    err = get_from_queue(&c, &val);
}
```

→ c->idx_prima_libera--; //ho una posizione libera in più
return OK;

}

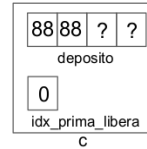
Politica di accesso

Possibile implementazione di una coda

```
#define OK 0
#define ERR 1
int get_from_queue(struct Coda* c,
                  int* p_val)
{
    if(c->idx_prima_libera == 0)
        return ERR;
    *p_val = c->deposito[0];

    int n_elementi = c->idx_prima_libera;

    //scorro a sinistra
    for(int i=0; i < n_elementi-1; i++)
    {
        c->deposito[i] = c->deposito[i+1];
    }
}
```



1
err
88
val

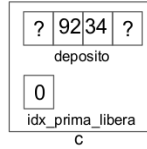
```
int main()
{
    struct Coda c;
    init_queue(&c);
    int err;
    err = insert_in_queue(&c, 42);
    if(err != 0)
        exit(-1);
    err = insert_in_queue(&c, 66);
    int val;
    err = get_from_queue(&c, &val);
    err = insert_in_queue(&c, 88);
    err = get_from_queue(&c, &val);
    err = get_from_queue(&c, &val);
    err = get_from_queue(&c, &val);
}
```

c->idx_prima_libera--; //ho una posizione libera in più
return OK;

}

Politica di accesso

Cose da NON fare



Una struttura dati che adotta una particolare politica di accesso deve quindi essere composta da:

- una struttura dati d'appoggio che conterrà materialmente i dati (esempio: un array)
- un insieme di funzioni ad-hoc che permettono di accedere alla struttura dati rispettando le regole della politica di accesso

L'accesso a tale struttura deve quindi essere effettuato unicamente tramite le funzioni definite.

Accedervi in qualsiasi altro modo è una violazione della politica d'accesso!

```
int main()
{
    struct Coda c;
    c.deposito[2] = 34; /* accesso diretto alle caratteristiche interne della struttura
                        senza passare per le funzioni specifiche!
                        VIOLAZIONE POLITICA FIFO
                        */
    c.deposito[1] = 92; // VIOLAZIONE POLITICA FIFO
    printf("%d\n", c.deposito[3]); // VIOLAZIONE POLITICA FIFO
}
```

Il C non mette a disposizione dei modi per evitare l'accesso a determinati campi di una struct. Quindi, se conosco i dettagli della struttura dati, posso praticamente accedervi come ho appena fatto senza passare per le funzioni ad-hoc. Ma ciò fa decadere la proprietà di essere una Coda alla struttura.

L'utilizzo delle sole funzioni ad-hoc per l'utilizzo della struttura mi garantisce che la struttura dati allocata sia effettivamente una Coda, e che quindi al suo interno troverò sempre i dati disposti nella maniera regolata dalla politica scelta.

Politica di accesso

Ricapitolando

Per definire una struttura dati con una certa politica d'accesso:

- definisco la struttura dati d'appoggio
- definisco le funzioni per accedervi

Le funzioni per accedervi possono a loro volta essere:

- fondamentali: richieste per l'accesso materiale alla struttura
- ausiliarie: non indispensabili, ma aiutano l'accesso alla struttura

Ad esempio, una migliore implementazione di una coda potrebbe prevedere le seguenti funzioni:

- fondamentali:

- `init_queue(...)`
- `insert_in_queue(...)`, o anche `enqueue(...)` o anche `append(...)`
- `get_from_queue(...)`, o anche `dequeue(...)`

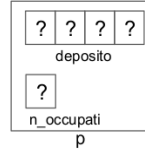
- ausiliarie:

- `is_queue_empty(...)`: restituisce un valore che indica se la coda è vuota o meno
- `is_queue_full(...)`: restituisce un valore che indica se la coda è piena o meno
- `print_queue(...)`: stampa il contenuto attuale della coda
- `read_from_queue(...)`: legge l'elemento in testa ma senza eliminarlo

Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
struct Pila
{
    int deposito[N_PILA];
    int n_occupati;
};
```



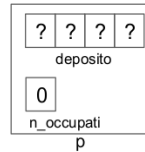
```
int main()
{
    struct Pila p;
}
```

Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
struct Pila
{
    int deposito[N_PILA];
    int n_occupati;
};

void init_stack (struct Pila* p)
{
    p->n_occupati = 0;
}
```



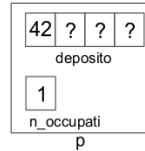
```
int main()
{
    struct Pila p;
    init_stack(&p);
}
```

Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0
```

```
int push(struct Pila* p, int val)
{
    if (p->n_occupati >= N_PILA)
        return FULL;
    p->deposito[p->n_occupati] = val;
    p->n_occupati++;
    return OK;
}
```



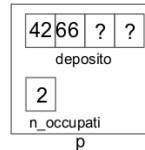
```
int main()
{
    struct Pila p;
    init_stack(&p);
    push(&p, 42);
}
```

Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0
```

```
int push(struct Pila* p, int val)
{
    if (p->n_occupati >= N_PILA)
        return FULL;
    p->deposito[p->n_occupati] = val;
    p->n_occupati++;
    return OK;
}
```



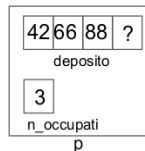
```
int main()
{
    struct Pila p;
    init_stack(&p);
    push(&p, 42);
    push(&p, 66);
}
```


Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0
```

```
int push(struct Pila* p, int val)
{
    if(p->n_occupati >= N_PILA)
        return FULL;
    p->deposito[p->n_occupati] = val;
    p->n_occupati++;
    return OK;
}
```



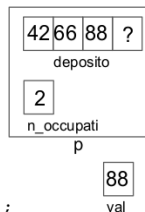
```
int main()
{
    struct Pila p;
    init_stack(&p);
    push(&p, 42);
    push(&p, 66);
    push(&p, 88);
}
```

Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0
```

```
int pop(struct Pila* p, int* val)
{
    if(p->n_occupati <= 0)
        return EMPTY;
    *val = p->deposito[p->n_occupati-1];
    p->n_occupati--;
    return OK;
}
```



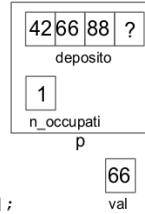
```
int main()
{
    struct Pila p;
    init_stack(&p);
    push(&p, 42);
    push(&p, 66);
    push(&p, 88);
    pop(&p, &val);
}
```

Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0
```

```
int pop(struct Pila* p, int* val)
{
    if(p->n_occupati <= 0)
        return EMPTY;
    *val = p->deposito[p->n_occupati-1];
    p->n_occupati--;
    return OK;
}
```



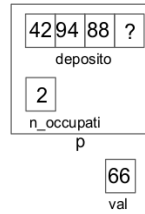
```
int main()
{
    struct Pila p;
    init_stack(&p);
    push(&p, 42);
    push(&p, 66);
    push(&p, 88);
    pop(&p, &val);
    pop(&p, &val);
}
```

Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0
```

```
int push(struct Pila* p, int val)
{
    if(p->n_occupati >= N_PILA)
        return FULL;
    p->deposito[p->n_occupati] = val;
    p->n_occupati++;
    return OK;
}
```



```
int main()
{
    struct Pila p;
    init_stack(&p);
    push(&p, 42);
    push(&p, 66);
    push(&p, 88);
    pop(&p, &val);
    pop(&p, &val);
    push(&p, 94);
}
```

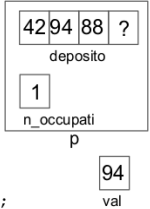
Politica di accesso

Possibile implementazione di una pila

```
#define N_PILA 4
#define FULL -1
#define EMPTY -2
#define OK 0

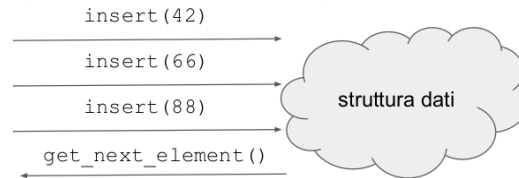
int pop(struct Pila* p, int* val)
{
    if(p->n_occupati <= 0)
        return EMPTY;
    *val = p->deposito[p->n_occupati-1];
    p->n_occupati--;
    return OK;
}

int main()
{
    struct Pila p;
    init_stack(&p);
    push(&p, 42);
    push(&p, 66);
    push(&p, 88);
    pop(&p, &val);
    pop(&p, &val);
    push(&p, 94);
    pop(&p, &val);
}
```



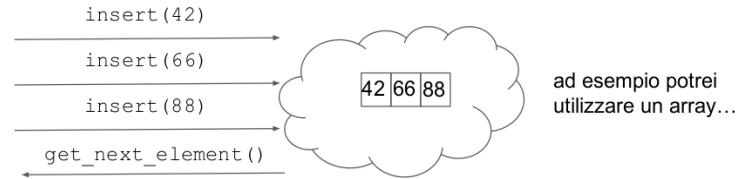
Politica di accesso

- L'utilizzo delle funzioni ad-hoc permette di evitare di dover conoscere (o ricordare) i dettagli implementativi della struttura.
- Mi è sufficiente conoscere i *prototipi* delle funzioni di accesso per poter accedere alla struttura, **senza dover sapere materialmente come è implementata la mia struttura dati.**



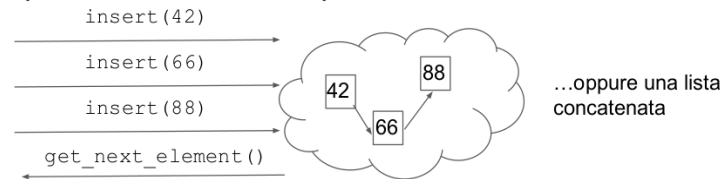
Politica di accesso

- L'utilizzo delle funzioni ad-hoc permette di evitare di dover conoscere (o ricordare) i dettagli implementativi della struttura.
- Mi è sufficiente conoscere i *prototipi* delle funzioni di accesso per poter accedere alla struttura, **senza dover sapere materialmente come è implementata la mia struttura dati.**



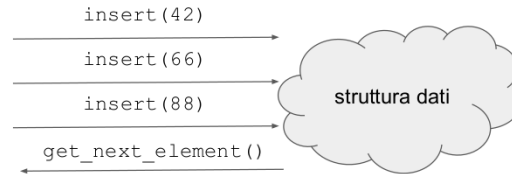
Politica di accesso

- L'utilizzo delle funzioni ad-hoc permette di evitare di dover conoscere (o ricordare) i dettagli implementativi della struttura.
- Mi è sufficiente conoscere i *prototipi* delle funzioni di accesso per poter accedere alla struttura, **senza dover sapere materialmente come è implementata la mia struttura dati.**



Politica di accesso

- L'utilizzo delle funzioni ad-hoc permette di evitare di dover conoscere (o ricordare) i dettagli implementativi della struttura.
- Mi è sufficiente conoscere i *prototipi* delle funzioni di accesso per poter accedere alla struttura, **senza dover sapere materialmente come è implementata la mia struttura dati.**



- qualsiasi sia la struttura dati materialmente implementata, per l'utilizzatore della struttura non ha alcuna importanza. Le funzioni di accesso nascondono i dettagli implementativi, ma permettono comunque l'accesso alla struttura da parte dell'utilizzatore.