



- È rilasciata in questo caso?

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    vector<int> v;
    // ...
    if (x) p[x] = v.at(x);    // at(): accesso con verifica
    // ...                      // può lanciare eccezioni
    delete[] p;
}
```

- È lanciata un'eccezione



# Gestione delle risorse

- È rilasciata in questo caso?

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    vector<int> v;
    // ...
    if (x) p[x] = v.at(x);    // at(): accesso con verifica
    // ...                      // può lanciare eccezioni
    delete[] p;
}
```

- È un problema di eccezioni o di gestione delle risorse?



- Una possibile soluzione se è un problema di eccezioni

```
void suspicious(int s, int x)
{
    int* p = new int[s];
    vector<int> v;
    // ...
    try {
        if (x) p[x] = v.at(x);
        // ...
    } catch (...) {
        delete[] p;
        throw;           // rilancia l'eccezione
    }
    // ...
    delete[] p;
}
```



## Soluzione 1

- Il codice precedente risolve il problema, però...
  - Il codice si duplica e diventa molto lungo
  - Se estendiamo questo approccio, dobbiamo gestire eccezioni:
    - A ogni accesso al vettore
    - A ogni allocazione
- Questo codice è difficile da leggere e da mantenere



## Soluzione 2

- Usando solo std::vector (al posto dell'allocazione dinamica) il problema è risolto

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    // ...
}
```

- Perché è risolto? Chi lo risolve e in che modo?



# Liberare risorse

- Il problema è risolto perché la memoria è acquisita in fase di costruzione e liberata in fase di distruzione
  - Costruttori e distruttore risolvono il problema
  - Lo risolvono in maniera **semplice**
- Quando un flusso di esecuzione (thread) lascia uno scope, sono **invocati i distruttori** di tutti gli oggetti e i sotto-oggetti



- Occupare le risorse nel costruttore e liberarle nel distruttore risolve questi problemi
  - L'oggetto è creato solo se è stato possibile acquisire le risorse
- Questa tecnica prende il nome di **Resource Acquisition Is Initialization (RAII)**
- È un'idea generale: vale anche per
  - socket
  - I/O buffer (inclusi i file)
  - ...

Agganciare l'acquisizione  
delle risorse al costruttore  
e il rilascio al distruttore



## RAll e scope

- Il meccanismo visto prima si basa sull'uscita dallo scope
  - L'uscita dallo scope libera le risorse mediante chiamata ai distruttori
  - Questo meccanismo permette di ottenere un buon design nella maggior parte dei casi
- Cosa succede se il flusso del codice deve uscire dallo scope per altri motivi, ma non per liberare le risorse?



# RAll e scope

```
vector<int>* make_vec()
{
    vector<int>* p = new vector<int>;
    // ... riempimento del vettore con i dati - può
    // ritornare un'eccezione!
    return p;
}
```

- A volte desideriamo "far uscire" gli oggetti dallo scope
  - Caso tipico: funzione che costruisce un oggetto grande e lo ritorna usando un puntatore



## RAll e scope

- Nota: allocare dinamicamente uno std::vector è solitamente **cattiva pratica**
  - Usato come esempio perché il riempimento può causare un'eccezione



# RAll e scope

```
vector<int>* make_vec()
{
    vector<int>* p = new vector<int>;
    // ... riempimento del vettore con i dati - può
    // ritornare un'eccezione!
    return p;
}
```

- Cosa succede se un'eccezione è lanciata durante il riempimento?
- Caveat: p deve essere liberato dal chiamante

Avviene un'allocazione dinamica della memoria, che è manuale, quindi la memoria non viene modificata fino alla chiamata della delete.  
Stiamo disattivando il meccanismo per cui le risorse venivano gestite correttamente.



## RAll e scope

- La funzione `make_vec` potrebbe dover gestire un'eccezione
- Comportamento desiderato:
  - Se non sono lanciate eccezioni, `make_vec` restituisce il puntatore
  - Se sono lanciate eccezioni, `return` non è eseguito, ma `make_vec` non deve comunque causare memory leak



# RAll e scope

- Implementazione di tale comportamento:

```
vector<int>* make_vec()
{
    vector<int>* p = new vector<int>;
    try {
        // ... riempimento del vettore con i dati - può
        // ritornare un'eccezione!
    }
    catch(...) {
        delete p;
        throw; // rilancia l'eccezione
    }
    return p;
}
```



# Garanzie

- La tecnica vista è un pattern ricorrente che prende il nome di basic guarantee
- **Basic guarantee:** il blocco try/catch fa sì che make\_vec() funziona, oppure lancia un'eccezione e non crea leak
  - Necessaria per tutto il codice che deve gestire eccezioni che potrebbero essere lanciate
  - STL fornisce la basic guarantee



# Garanzie

- **Strong guarantee:** la funzione rispetta la basic guarantee, e in più tutti i valori osservabili (valori non locali alle funzioni) sono gli stessi che erano presenti prima della chiamata **in caso di fallimento**
  - Commit or rollback
- **No-throw guarantee:** la funzione non lancia eccezioni



# Garanzie di make\_vec()

- Che garanzia offre make\_vec()?

```
vector<int>* make_vec()
{
    vector<int>* p = new vector<int>;
    try {
        // ... riempimento del vettore con i dati - può
        // ritornare un'eccezione!
    }
    catch(...) {
        delete p;
        throw; // rilancia l'eccezione
    }
    return p;
}
```

Se c'è gestione delle eccezioni, in generale non c'è la no throw guarantee



# Garanzie di make\_vec()

- Che garanzia offre make\_vec()?
  - Sicuramente la basic
  - Anche la strong, se non ci sono istruzioni strane nel riempimento del vettore
- La funzione make\_vec() funziona, ma c'è un modo per evitare il try/catch?
  - È "brutto": ci obbliga a scrivere software per gestire il caso particolare

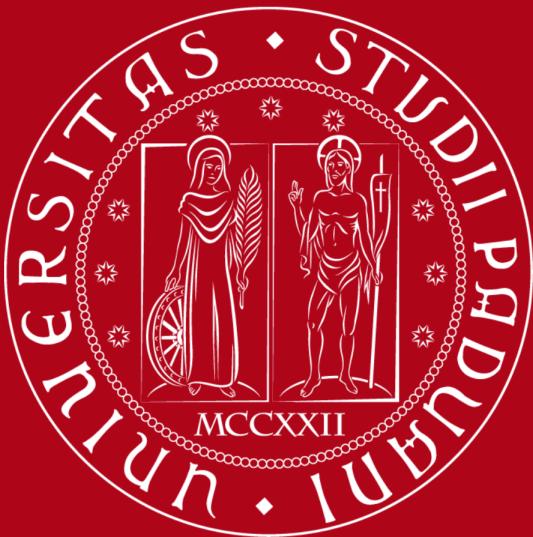


# Uscita dallo scope e deallocazione

- Risolviamo il problema precedente se riusciamo a collegare:
  - Uscita dallo scope
  - Deallocazione
- Esistono strumenti che gestiscono questa situazione: **smart pointer**



- Gestire le risorse in presenza di eccezioni
- RAII
- Garanzie
  - Basic guarantee
  - Strong guarantee
  - No-throw guarantee



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

## Risorse ed eccezioni

Stefano Ghidoni