Domanda 1

Le eccezioni sono un meccanismo tipico della OOP che permette di intercettare errori che potrebbero compromettere il flusso di esecuzione del codice e gestirli.

La necessità di un meccanismo di questo tipo è nata al crescere della dimensione e complessità del software, che ha reso più probabile il verificarsi di errori di "basso livello" (gestione memoria, ad esempio) e, al contempo, più difficile e dispendioso individuarli. Le eccezioni permettono di identificare dei blocchi problematici già a tempo di scrittura del codice; se l'errore si verifica, si dice che viene "sollevata un'eccezione" e viene lanciato il blocco di codice che la gestisce.

Sintassi di un blocco `try-catch`:

```
try {
    ...
    if (<casino>)                    // se succede un casino
        throw "è successo un casino"; // lancia l'eccezione (in questo caso una string)
    ...
} catch (<eccezione>) {
    // Gestisci il casino
}
```

L'eccezione può anche essere sollevata all'interno della chiamata di una funzione e essere gestita da un blocco catch nel chiamante; più in generale, le eccezioni vengono propagate e risalgono lo stack delle hiamate.

Un potenziale problema legato a questo meccanismo può presentarsi quando, nel nostro codice, andiamo a importare numerose librerie, perché queste potrebbero avere numerose eccezioni e blocchi `catch` al loro interno; dovessero occorrere degli errori, intercettati e gestiti dalle eccezioni già presenti nelle librerie, sarebbe molto difficile per il programmatore andare a scoprire cosa è andato storto e dove.

D'altro canto potrebbe anche verificarsi il contrario, ovvero se importiamo una libreria che lancia eccezioni non gestite a nostra insaputa il nostro programma potrebbe bloccarsi in modi inaspettati.

Ecco alcune tra le eccezioni più comuni:

- `throw "casino"; catch (const char* msg)`: semplice stringa
- `throw invalid_argument("valore negativo")`
- `throw overflow_error("divisione b = 0!");`
- `throw out_of_range("a o b maggiori di 9");`
- `throw logic_error("grosso guaio");`
- `throw bad_alloc("memoria finita");`
- `catch (exception& e)`: cattura l'eccezione, si può stampare con `cout << e.what();`
- `catch (...)`: si comporta più o meno come il default di uno switch statement

Domanda 2

La libreria `mutex` è stata introdotta come soluzione a basso livello per la gestione del cosiddetto *data race* problem.

Questo è un problema che si verifica quando lavoriamo in multithreading e abbiamo due processi separati che vanno a effettuare delle operazioni non atomiche su una stessa variabile, che ci espone al rischio concreto di ottenere inconsistenze nell'elaborazione della stessa, dal momento che non è possibile controllare l'operato dello scheduler che decide quale processo mandare in essecuzione.

Il funzionamento è quello di bloccare l'accesso alla cosiddetta sezione critica, ossia quella porzione di codice che comprende le operazioni da effettuare sulla variabile incriminata, e rilasciarla alla fine delle stesse; le istanze di mutex effettuano queste operazioni utilizzando, rispettivamente, i metodi `lock()` e `unlock()`.

Tuttavia, non è una panacea: come si evince, mutex ricchiede di essere gestita dal programmatore, il quale potrebbe benissimo dimenticarsi di rilasciare il lock e, di fatto, bloccare il flusso di esecuzione per tutti quei sotto processi che interverranno su quella variabile succcessivamente.

Inoltre mutex lascia irrisolti anche altri problemi, come *deadlock* e *starvation*.

Domanda 3

Un move constructor è un costruttore che riceve come parametro un rvalue reference di un'istanza della sua classe e ne crea una nuova copiando in quella nuova i campi statici e (soprattutto) acquisendo l'ownership dei puntatori, lasciando infine l'istanza passata in uno stato valido (ossia parametri statici consumati e parametri dinamici non inizializzati).

Profondamente diverso da un copy constructor, che invece, a partire da un const lvalue reference di un'istanza, ne crea una nuova andando a dichiarare un puntatore e riallocare un buffer di memoria dedicato, in cui va a duplicare il contenuto referenziato dal puntatore dell'istanza passata (operazione che ovviamente va eseguita per tutti i campi dinamici).

Esempi:

```
1  class CC
2  {
3      int* pip;
4  public:
5      // MOVE CONSTRUCTOR
6      A (A&& other)
7      {
8          this->pip = other.pip;
9          other.pip = nullptr;
10     }
11
12     // COPY CONSTRUCOR
13     A (const A& other)
14     {
15         if (other.pip != nullptr)
16             this->pip = new int(*(other.pip));
17         else
18             this->pip  = nullptr;
```

```
    }
}
```

Domanda 4

Una lambda expression è un `prvalue` il cui oggetto risultato è chiamato *oggetto chiusura*; nella pratica, forniscono una sintassi concisa per creare funzioni senza nome.

Risultano particolarmente comode quando si ha da passare delle semplici funzioni “al volo” come argomento passato ad altre funzioni; impossibile non menzionare l’uso per le funzioni della libreria `algorithm` come esempio.

Struttura (`mutable` e il ritorno sono opzionali):

```
[captures] (params) mutable → ret { body }
```

La cattura è un meccanismo per utilizzare delle variabili esterne all’interno dello scope della lambda. Abbiamo due tipi di cattura: per riferimento e per copia. Il tipo di cattura può essere specificato anteponendo il simbolo apposito prima di ogni variabile, oppure una volta sola come primo elemento del blocco di cattura. Nel dettaglio, i tipi di passaggio sono tali che:

- copia: è di default, ma si può esplicitare con il simbolo `=`; non permette la modifica dei parametri catturati,
  che però può essere attivata con la keyword `mutable` (lal modifica permane solo all’interno dello scope);
- riferimento: va specificata con `&` e, come ci si aspetta, i cambiamenti fatti ai parametri passati in questo modo
  permangono anche fuori dallo scope.

Un esempio d’uso:

```
list<int> l;
int accumulator = 0;
for_each(l.begin(), l.end(), [&accumulator] (int n) {
    accumulator += n;
});
```