Domanda 1

Gli *smart pointers* sono delle classi della STL che servono per implementare dei puntatori "intelligenti", ovvero dei puntatori che aiutano il programmatore a gestire la memoria assolvendo dei compiti come la deallocazione della memoria in automatico.

Esistono i seguenti tipi di smart pointer:

- *unique pointer*: puntatore che può essere posseduto da una sola istanza di oggetto, viene utilizzato per puntare un oggetto che deve poter essere posseduto da un unico proprietario; la responsabilità di gestione della memoria allocata per lo unique pointer è dell'unico oggetto possessore del puntatore;
- *shared pointer*: al contrario dello unique pointer, questo puntatore può essere posseduto da diversi oggetti, la classe shared pointer contiene al suo interno le informazioni per registrare quali e quanti oggetti posseggono il puntatore; la responsabilità della deallocazione spetta all'ultimo oggetto rimanente con una copia dello shared pointer.
- *weak pointer*: condivide una risorsa ma non ne è il proprietario (e di conseguenza non aumenta il valore dello `use_count()`) e non è nemmeno dereferenziabile: per fare questo bisogna copiarlo in uno shared pointer utilizzando il metodo `lock()`; l'uso di questo metodo, inoltre, blocca la modifica del puntatore da parte degli altri shared pointers che lo condividono fino a quando questo weak non viene rilasciato.

Domanda 2

L'ereditarietà mutipla è un meccanismo che permette ad una classe di ereditare da più classi base differenti. Consideriamo una classe `A` che eredita da due classi `B` e `C` e implementiamo:

```
class A
    : public B
    , public C
{
    ...
}
```

Come si vede, stiamo ereditando in maniera `public` da `B` e `C`, in quest'ordine.

Il *diamond problem* si verifica quando le due classi padri `B` e `C` ereditano esse stesse da una stessa classe `D`. Se `D` contiene un attributo `protected` o `private` questo verrà ereditato sia da `B` che da `C`; come faremo a decidere quale versione di tale attributo deve essere ereditata da `A`? La risposta è semplice: si deve fare in modo che `B` e `C` ereditino da `D` in modo virtuale.

Inoltre, l'ereditarietà virtuale risolve anche il problema di chiamate multiple al costruttore della classe `D`.

Domanda 3

Le eccezioni in `C++` sono un meccanismo di segnalazione degli errori che permette di regolare il comportamento di un programma nel caso si verifichino degli errori inaspettati a runtime. È utile nel caso in cui un programma sia molto delicato e non si vuole che questo venga interrotto senza prima compiere delle operazioni di messa in sicurezza del sistema.

Le eccezioni possono essere lanciate dal programma quando si verificano determinate condizioni per cui è stato predisposto il lancio di queste; queste possono poi essere "catturate" tramite l'impiego di blocchi `catch` che appunto contengono il codice da eseguire una volta che si verifica una determinata eccezione.

La sintassi completa è la seguente:

```
try {
    // qui vanno le operazioni che potrebbero causare
    // l'eccezione e le operazioni di lancio eccezione
} catch (<tipo di eccezione>) {
    // qui le operazioni da svolgere nel caso
    // in cui si verifichi l'eccezione
}
```

Domanda 4

I thread sono un modo per introdurre la multiprogrammazione nel `C++`: danno la possibilità al programmatore di definire delle funzioni e lanciarle come processi separati, in modo da permettere a macchine multi-core di svolgere contemporaneamente queste funzioni, velocizzando di molto il tempo di esecuzione dei programmi.

Il problema conosciuto come *data-race* è un problema introdotto appunto dalla possibilità di avere thread, che sono processi che condividono lo stesso spazio di indirizzamento, e che quindi possono agire sulle stesse locazioni di memoria in modo non regolato dal punto di vista dell'ordine degli accessi: di fatto non possiamo sapere come lo scheduler dei processi ordinerà lo svolgimento dei vari thread, quindi se utilizziamo la stessa variabile su thread diversi dobbiamo aspettarci che questa possa essere modificata con ordine imprevedibile dai due thread; in sostanza, un caso di data race non trattato ci fa rischiare di lavorare con dati non consistent.

Per risolvere questo problema si devono usare delle strategie di regolamentazione di accesso alle variabili nello spazio di indirizzamento comune. Ecco alcune possibilità:

- le variabili possono essere dichiarate con delle librerie di supporto come `atomic` che offrono delle classi wrapper per gestire la sincronizzazione dell'accesso alla memoria;
- l'accesso alle variabili può essere regolato con librerie di supporto come `mutex`;
- l'accesso alle variabili può essere regolato, a livello abbastanza basso, tramite l'impiego di semafori (che alla fine sono implementati in `mutex` in maniera più semplice).