Domanda 1

L'ereditarietà è un meccanismo tipico della OOP per cui è possibile definire delle classi, dette derivate, a partire dalla definizione di altre classi base. Le classi derivate condivideranno parte (o anche tutti) gli attributi delle classi base, a cui se ne possono aggiungere eventualmente degli altri.

In C++ abbiamo tre tipi di ereditarietà (si consideri una classe Base `A` e una derivata `B`):

- `public`: tutti gli attributi definiti nell'area `public` di `A` resteranno `public` in `B`, analogamente per quelli definiti nell'area `protected`
- `protected`: tutti gli attributi che `B` eredita da `A` risulteranno `protected` in `B`
- `private`: tutti gli attributi che `B` eredita da `A` risulteranno `private` in `B`

Alcuni esempi:

```
class A {
private:
    int a;
protected:
    int b, c;
public:
    A (int _a, int _b) { ... };
    ...
}

class B
    : public A
{
private:
    // avrò a disposizione dei campi b, c
    int d;
public:
    ...
}
```

Ereditare con `public` fa sì che la relazione tra derivata e base sia una IS-A, per cui ad esempio posso usare un puntatore del tipo della classe base per riferirmi a un'istanza della derivata.

```
int main() {
    A* a = new B(); // ok
}
```

Tuttavia questo potrebbe causare dei problemi se andiamo nella classe figlia andiamo a fare l'overriding di alcuni metodi. Per risolvere il problema e forzare il late binding (quindi la ricerca di quale metodo usare a runtime, cfr. nell'early binding questo è risolto a compile time), possiamo anteporre la keyword `virtual` ai metodi che ci interessano.

Possiamo anche definire un metodo come *puramente virtuale* in questo modo:

```
class foo {
    ...
    virtual void tritura(int media) = 0;
}
```

La presenza di un metodo puramente virtuale renderà la classe puramente virtuale a sua volta, vale a dire non instanziabile e utilizzabile solo per derivare delle classi figlie.

Domanda 2

Un *lvalue* è un oggetto che ha un preciso indirizzo in memoria ed è referenziabile; un *rvalue* è un oggetto temporaneo che non possiede un indirizzo di memoria e non è referenziabile, solitamente delle costanti letterali, ad esempio (come numeri, `true`, `null`). Sono detti così perché gli *lvalue* possono stare sia a sinistra che a destra di `=`, anche se solitamente li si trova più spesso a sinistra; di contro, gli *rvalue* possono trovarsi solamente a destra dell'uguale.

```
    int a = 3; // assegnamento lvalue = rvalue, ok
    int b = a; // lvalue = lvalue, ok
    3 = a; // inutile specificare che sta roba fa strabuzzare gli occhi
```

Per tutti questi motivi, se abbiamo una funzione che accetta dei riferimenti, non sarà possibile passarle un rvalue.

```
int foo (int& n) {
    // non mandare i tuoi sogni in foomo
}

int main() {
    int a = 4;
    foo(a); // lvalue, ok
    foo(4); // rvalue, NON COMPILA
}
```

Esiste un modo per far funzionare tutto anche senza rvalue reference: possiamo dichiarare il parametro come `const` nella firma della funzione.

```
int foo (const int& n) {
    // foonghi
}

int main() {
    int a = 4;
    foo(a); // lvalue, ok
    foo(4); // adesso funziona
}
```

Domanda 3

Gli iteratori sono delle classi template che fanno l'overloading di alcuni operatori secondo il comportamento che questi hanno nell'aritmetica dei puntatori, di fatto surrogandone il funzionamento e rendendolo conveniente all'uso delle classi della STL. Esiste una gerarchia: *Random access* > *Bidirectional* > *Forfard* > *Input* | *Output*; a seconda del contenitore della STL in questione, i suoi iteratori staranno a diversi livelli della gerarchia (una lista monodirezionale non può avere un iteratore bidirezionale).

Gli iteratori possono essere anche di altri tipi: possono essere *const* (non consentono di modificare il valore puntato), *reverse* (`begin()` e `end()` sono invertiti), o ovviamente anche questi ultimi assieme.

```
int main() {
    vector<int> v = {1,2,3,4,3,41,3,6,2};

    vector<int>::iterator it;
    for (it = v.begin(); it =/= v.end(); ++it) {
        cout << *it << endl; // stampa v
    }

    vector<int>::const_iterator cit;
    for (it = v.cbegin(); it =/= v.cend(); ++it) {
        cout << *it << endl; // stampa v, non modifica i valori
    }

    vector<int>::reverse_iterator rit;
    for (it = v.rbegin(); it =/= v.rend(); ++it) {
        cout << *it << endl; // stampa v al contrario
    }

    vector<int>::const_reverse_iterator crit;
    for (it = v.crbegin(); it =/= v.crend(); ++it) {
        cout << *it << endl; // stampa v al contrario, non modifica
    }
}
```

Domanda 4

Il multithreading è un meccanismo che consente di trarre vantaggio dai processori di architettura multicore facendo in modo che alcune funzioni vadano eseguite come processi separati (thread) in parallelo su diversi core. Se usato sapientemente può fornire un boost di prestazioni ed efficienza decisamente non trascurabile, ma pone un numero di sfide maggiore per il programmatore.

Uno dei problemi più noti è il cosiddetto *data race* problem: dal momento che i processi nei diversi thread condividono lo stesso spazio di indirizzamento, potrebbero verificarsi delle inconsistenze nella scrittura/lettura di alcune variabili, se le operazioni su di esse non sono adeguatamente controllate. Esistono diverse soluzioni:

- si può fare affidamento alla libreria atomic;
- si può utilizzare mutex;
- semafori.

Altri problemi sono ad esempio il cosiddetto deadlock, ovvero una situazione in cui un gruppo di processi è in attesa del rilascio di una variabile che è posseduta da uno di loro e che non potrà mai essere rilasciata, o la starvation.