Domanda 1

Il concetto di *copia profonda*, nell'ambito della OOP, si contrappone a quello di *shallow copy*. Quest'ultimo è un meccanismo di copia in cui il valore di ogni campo dell'istanza di una certa classe viene copiato in una nuova istanza della stessa classe, ma questo causa inevitabilmente dei problemi in cui uno dei membri non sia un'oggetto allocato staticamente nello stack, ma bensì dinamicamente nell'heap, come ad esempio:

```
class Referendum
{
    int sì;
    int no;
    char* quesito;
};
```

Il problema è che se copiassimo uno a uno questi campi da un'istanza `r1` in una nuova istanza `r2` finiremmo per duplicare semplicemente il *puntatore* a `quesito`, per cui avremmo che `r1.quesito` e `r2.quesito`, pur appartenendo a istanze diverse della stessa classe, punteranno allo stesso oggetto nella memoria heap, e questo è evidentemente poco desiderabile se il motivo per cui abbiamo fatto la copia è che vogliamo avere due istanze di `Referendum` diverse e indipendenti l'una dall'altra (aka modificare qualsiasi campo dell'una non altera in alcun modo lo stato dall'altra).

Per ovviare a questo problema dobbiamo ridefinire costruttore di copia e operatore di assegnamento in modo tale che, in una loro chiamata, venga allocata nell'heap della nuova memoria in cui copiare l'oggetto puntato da tutti i membri puntatore della classe in questione.

```
class A
{
  private:
    int i;
    B* bp;
  public:
    A()
    {
        i = 0;
        bp = NULL;
    }

    // costruttore copia profonda
    A(const A& other)
    {
        i = other.i;
        if other.bp =/= NULL
            bp = new B(*(other.bp));
        else
            bp = NULL;
    }

```

```
    // operatore assegnazione
    A& operator = (const A& other)
    {
        if (this->bp == NULL) {
            if (other.bp != NULL)
                bp = new B(*(other.bp));
        } else {
            delete bp;
            if (other.bp != NULL) {
                (*bp) = *(other.bp);
            }
            else {
                bp = NULL;
            }
        }
    }
}
```

Domanda 2

La programmazione generica è un meccanismo di `C++` che permette di definire classi o funzioni parametrizzando il tipo dei parametri a tempo di compilazione.

Si tratta di uno strumento estremamente comodo dal punto di vista di mantenibilità e agilità di scrittura del codice, perché permette di scrivere una volta sola del codice che altrimenti andrebbe ripetuto uguale (o quasi) per diversi tipi di dato; è quindi un ottimo meccanismo per introdurre astrazione nel codice.

Abbiamo due tipi di template:

- funzioni, si usano con `template <typename T>`
- classi, si usano con `template <class T>`

Quando si utilizzano i template è consigliablie scrivere l'implementazione di metodi e funzioni nei file `.h`, non perché farlo in un `.cpp` esterno non sia possibile, ma semplicemente perché è complicato fuori maniera e per nessuna ragione.

Un esempio delle potenzialità di questo astuto ingegno è la *Standard Template Library*, che fornisce dei contenitori (strutture dati) con metodi già pronti e che appunto parametrizzano il tipo di dato.

Un piccolo esempio di funzione template, supponendo di avere già implementato l'overload dell'operatore `+` per la classe `T`, nel caso in cui `T` fosse una classe:

```
template <typename T>
T superSomma(T& t1, T& t2) const {
    return T(t1 + t2);
}
```

Domanda 3

Il contenitore `set` della STL è una struttura template che:

- è organizzata come un binary search tree, per cui la ricerca di un elemento è $S(\log n) = O(\log n)$ e inserimento e rimozione sono $O(\log n)$;
- non possiede i concetti di testa e coda, in quanto BST;
- non può possedere elementi duplicati al suo interno;
- non si possono modificare il valore di elementi già presenti;
- gli elementi sono ordinati .

Per quest'ultimo motivo è necessario che il tipo di dato con cui viene parametrizzato il set abbia *sempre l'operatore* `<` *definito*, altrimenti non sarebbe possibile ordinare gli elementi in fase d'inserimento (e quindi lavorare con un BST).

Sono utilizzabili tutti e quattro gli iteratori (combinazioni di `const` e `reverse`).

Domanda 4

Un move constructor è un costruttore che riceve in input un rvalue reference di un'istanza della classe di riferimento e permette che una nuova istanza venga inizializzata "rubando" i campi valorizzati di quella passata (che pur resta in uno stato valido), ossia impossessandosi dei suoi puntatori, senza quindi che vengano effettuate copie non necessarie.

```
1  class A
2  {
3    private:
4      int i;
5      string* s;
6
7    public:
8      A (int _i, string* _s) { ... };
9
10     // copy construtor (deep copy)
11     A (const A& other)
12     {
13         i = other.i;
14         if (other.s =/= NULL)
15             s = new string(*(other.s));
16         else
17             s = NULL
18     }
19
20     // move constructor
21     // argument is a rvalue reference,
22     A (A&& other)
23     {
24         // mi impossesso del puntatore dell'altro
25         s = other.s;
26         // lasciamo other in uno stato valido
27         // vogliamo evitare che, ad esempio, il gc tenti
28         // di eliminare due volte questo stesso buffer di memoria
29         other.s = NULL;
30
```

```
        i = other.i;
        other.i = 0; // consumo il parametro dell'altro
    }
}
```

Per ottenere un rvalue reference posso utilizzare la funzione `std::move()`, che per l'appunto restituisce un rvalue reference dell'oggetto passato.

```
int main ()
{

    A a1(0, "feston");
    A a2(std::move(a1));

    return 0;
}
```