Domanda 1

Un metodo si definisce astratto (o, per dirla in maniera più propria per `C++`, *virtuale*) quando ne viene forzato il late binding anteponendogli alla firma la keyword `virtual`.

Si forza il late binding nelle situazioni in cui facciamo uso di ereditarietà e si corre il rischio che, istanziando una classe a runtime mediante la dichiarazione di un puntatore alla classe padre, si richiami un metoodo pensando di invocare quello della classe figlia (classe effettiva a runtime) ma so vada in effetti ad usare quello della classe padre (classe tatica riconosciuta a compile time); per l'appunto, risolvendo il binding del metodo a runtime si risolve questo problema.

```
class A
{
    ...
public:
    ...
    virtual void daSovrascrivere(int n) { ... }
    ...
}

class B
    : public A
{
    ...
public:
    ...
    void daSovrascrivere (int n) { ... }
    ...
}

int main() {
    A bp = new B();
    bp.daSovrascrivere(1); // senza virtual andremmo a richiamare
                           // il metodo di A::daSovrascrivere,
                           // mentre ora usiamo B::daSovrascrivere
}
```

Inoltre, mediante una precisa sintassi, la keyword `virtual` può essere usata per creare un metodo *puramente virtuale*, ossia privo di implementazione. La presenza di un solo metodo puramente virtuale è sufficiente a rendere la sua classe puramente virtuale a sua volta e pertanto non istanziabile; può solo essere usata come classe padre per altre derivate, che dovranno necessariamente fornire un'implementazione per tutti i metodi puramente virtuali del padre per essere instanziabili.

```
class Virt
{
    ...
public:
    ...
```

```
    virtual void puramenteVirtuale (int n) = 0; // quell'= 0 rende il metodo puramente virtuale
    ...
}

class Child
    : public Virt
{
    ...
public:
    ...
    void puramenteVirtuale (int n)
    {
        // non è più puramente virtuale qui ofc
    }
    ...
}
```

Domanda 2

Un *copy constructor* è un costruttore usato per effettuare la copia profonda di un'istanza di una certa classe a partire da un lvalue reference passato.

Per copia profonda intendiamo un meccanismo di copia in cui si a a creare una nuova istanza di una certa classe del tutto indipendente da quella passata per effettuare la copia; questo vuol dire che, se la classe presenta degli attributi di tipo puntatore, la copia profonda non copierà semplicemente il valore del puntatore (facendo sì che sia l'istanza passata che quella nuova puntino allo stesso oggetto, con grandi problemi di inconsistenza e deallocazione), andrà ad allocare della nuova memoria e ne copiarà il buffer puntato dal puntatore dell'istanza passata.

```
class A
{
private:
    int a;
    B* pb;
public:
    ...
    A(const A& _a)
    {
        i = _a.i;
        if (_a.pb =/= NULL){
            pb = new B(*(_a.pb));
        } else {
            pb = NULL;
        }
    }
    ...
}
```

Un *move contructor* è un costruttore che "trasferisce" i campi valorizzati dell'istanza di una certa classe, passata come rvalue reference, a una nuova istanza, lasciando l'istanza di partenza in uno stato valido ma non specificato. Questo meccanismo è utile quando si vogliono evitare la creazione di innumerevoli oggetti temporanei che vengono distrutti subito nei passaggi per

copia.

Il funzionamento prevede che gli attributi salvati nello stack siano copiati ed il loro valore venga "consumato" (settato a 0, o a una stringa vuota) nell'rvalue passato, mentre invece per i campi allocati dinamicamente si copia direttamente il puntatore dell'rvalue nella nuova istanza, e quindi si assegna `nullptr` al puntatore dell'rvalue (sempre per consumarlo).

```
class A
{
private:
    int a;
    B* pb;
public:
    ...
    A(A&& _a)
    {
        a = _a.i;
        _a.i = 0; // consumo il parametro statico

        pb = _a.pb; // rubo il puntatore dell'istanza passata
        _a.pb = nullptr; // lascio il puntatore in uno stato valido, ma ne consumo
                         // il valore  per evitare che la memoria sia cancellata
                         // da delle delete
    }
    ...
}
```

Domanda 3

Fare l'*overload* di un metodo significa creare molteplici metodi aventi lo stesso nome ma con delle differenze rispetto a numero e tipo di parametri passati (e, verosimilmente, implementazione).

Data una classe base che possiede un certo metodo (`public` o `protected`), si dice che se ne fa l'*override* quando, in una classe figlia della stessa, si va a definire un metodo avente uguale firma (quindi nome e numero/tipo di parametri) di quello nella classe base, di fatto "sovrascrivendolo".

```
class A
{
    ...
public:
    ...
    // un po' di overloading
    int magnificentCalculation(int n);
    int magnificentCalculation(long n);
    int magnificentCalculation(float n);
    int magnificentCalculation(double n);
    void magnificentCalculation(double n);
    ...
}
```

```
14
15  class B
16      : public A
17  {
18      ...
19  public:
20      ...
21      // stiamo facendo l'override del primo
22      // magnificentCalculation
23      int magnificentCalculation(int n);
24      ...
25  }
```

Domanda 4

Un puntatore a funzione è, come espresso dal nome stesso, un puntatore che referenzia una funzione. Nella dichiarazione bisogna specificare tipo di ritorno e tipo dei parametri, ovviamente. La sintassi per la dichiarazione e l'assegnamento a una funzione è questa:

```
1  // le parentesi attorno a *pfunz sono importanti:
2  // la loro assenza farebbe sì che l'operatore
3  // di deferenziazione venga associato al tipo di ritorno, e non a pfunz
4  int (*pfunz)(double, int);
5  int f1(double, int);
6  int f2(double, int);
7  ...
8  pfunz = f1; // o pfunz = f2, come preferite
```

Quando viene chiamato, l'uso dell'operatore di referenziazione è opzionale:

```
1  cout << (*pfunz)(2.0, 1) << endl; // ok
2  cout << pfunz(2.0, 1) << endl; // va bene uguale
```

Dal momento che quello di *puntatore a funzione* è effettivamente un tipo di dato possiamo farci un certo numero di operazioni potenzialmente interessanti. Ad esempio, possiamo utilizzare l'operatore `typedef`:

```
1  typedef double (*P)(double);
```

Posso anche realizzare array (o qualsiasi tipo di struttura dati) di puntatori a funzione:

```
1  map<P, double> m;
2  m[f1] = 2;
3  m[f2] = 4;
```

Oppure ancora posso utilizzare un puntatore a funzione come parametro passato a un'altra funzione, aprendo la strada a numerose possibilità di aumento di modularità e astrazione; si pensi, ad esempio, a un uso congiunto con le *lambda expressions*.