Domanda 1

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 }
```

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:

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

Questi due elementi sembrano evidentemente quasi complementari: si pensi, ad esempio, Alla possibilità di definire un puntatore a funzione come parametro nella firma di una funzione, andando così a fornire un elemento di astrazione (in modo simile a quanto avviene con le funzioni della STL), dal momento che quel puntatore a funzione può essere valorizzato a chiamata con una lambda dedicata.

Domanda 2

Abbiamo tre tipi di ereditarietà in `C++` ; nello specifico, quelli richiesti sono:

- `protected` : tutti gli attributi `protected` e `public` della classe base verranno ereditati come `protected` nella figlia;
- `private` : tutti gli attributi non privati della classe base verranno ereditati come `private` nella classe figlia.

```
class A
{
private:
    int priv;
protected:
    int protect;
public:
    A () { ... }
```

```
}

class B
    : protected A
{
protected:
    // in quest'area troviamo sia la variabile protect
    // sia il costruttore pubblico
}

class C
    : private A
{
    // tutti i campi di risultano private
}
```

Domanda 3

In `C++` abbiamo i concetti di lvalue e rvalue: i primi possiedono un indirizzo in memoria e possono trovarsi sia a sinistra che a destra di un operatore di assegnazione, i secondi invece sono dei valori che non possiedono indirizzo in memoria (solitamente costanti letterali o valori temporanei) e possono trovarsi solo a destra dell'uguale.

Esiste anche il concetto di lvalue reference, ossia un riferimento a un lvalue.

```
int a = 5;
int& b = a; // b è un lvalue reference
```

Se proviamo ad utilizzare un rvalue come argomento di una funzione che accetta lvalue reference otteniamo un errore, a meno che questo non sia passato con `const`.

```
void NonFunz(int& a) {
    // roba
}

void Funz(const int& a) {
    // roba che però funziona
}

int main() {
    int value = 5;
    NonFunz(value); // lvalue, no problem
    NonFunz(5);     // esplode tutto
    Funz(5);        // nessun problema, perché c'è const
}
```

In `C++11` è stato introdotto il concetto di rvalue reference, ossia un riferimento a un rvalue. Un rvalue reference è un entità dereferenziabile ed è quindi tecnicamente un lvalue.

```
int&& i = 7;
```

Nell'ambito della move semantics questo assume grande rilevanza, dal momento che gli rvalue reference sono utilizzati dai cosiddetti move constructors per creare nuove istanze di determinate classi a partire da degli oggetti temporanei (che sarebbero quindi rvalue e non potrebbero essere utilizzati in un costruttore, normalmente): utilizzando la funzione std::move()`, infatti, possiamo ottenere un rvalue reference dell'istanza passata e utilizzarlo nel move constructor.

```
int main(){
    A a1, a2(1);
    a1 = a2; // copia
    a1 = std::move(a2); // move constructor
}
```

Domanda 4

La libreria `algorithm` fornisce delle funzioni pensate per interagire con le classi template della STL e fornire degli algoritmi con il maggior grado di astrazione possibile rispetto al contenitore.

Per fare questo alle funzioni di `algorithm` non si passa il contenitore, bensì i suoi iteratori, che a loro volta sono delle classi template che ridefiniscono alcuni operatori con il funzionamento che questi hanno nell'aritmetica dei puntatori, costituendone di fatto un surrogato per i contenitori. Tramite gli iteratori, è possibile appunto specificare il range del contenitore su cui andare a effettuare l'operazione desiderata.

```
list<int> l; // si supponga popolata di valori casuali
list<int>::iterator it = ++(++(++(l.begin()))); // ottengo iteratore che punta al terzo elemento di l

// scorro i primi tre elementi e ne resituisco il quadrato
// sovrascrivendo la lista stessa a partire dal primo elemento
transform(l.begin(), it, l.begin(), [] (int n) {
    return n*n;
});
```