Domanda 1

Le lambda expression sono funzioni senza nome. Sostanzialmente questo significa che il paradigma delle lambda expression offre, nel `C++`, un modo per creare funzioni on-the-fly che vengono utilizzate spesso quando si deve creare al volo una piccola funzione (il classico esempio prevede che una lambda sia passato come parametro-funzione per la funzione `algorithm::transform(…)`).

Le lambda, tuttavia, offrono un meccanismo particolare molto interessante che le funzioni "normali" non offrono: la cattura. La forma di una lambda expression è la seguente:

```
[ <parametri catturati> ]( <parametri della funzione> ) {
corpo della funzione
};
```

L'osservatore più attento avrà notato che tra le parentesi quadre si vede appunto una zona chiamata parametri catturati; qui vanno posti proprio i nomi di quelle variabili che vogliamo vengano catturate dalla nostra lambda, ovvero variabili che sono visibili nello stesso scope della lambda e che, una volta passate per cattura, la lambda può utilizzare al suo interno. Questo meccanismo offre due modalità di cattura: per copia (normale) o per riferimento (con l'apposizione della canonica `&` prima del nome della variabile da catturare), sai già cosa significa.

Altre cose notevoli:

- di per sé, il contenuto di una variabile catturata per copia non può essere modificato, nemmeno localmente nello scope della lambda; se vogliamo che questo sia possibile, possiamo aggiungere la keyword `mutable` a seguito delle parentesi tonde;
- una lambda expression può essere assegnata a una variable dichiarata con la keyword `auto`;
- il tipo di ritorno di una lambda può essere tipizzato esplicitamente dal programmatore (altrimenti verrà fatto in automatico dal compilatore);
  ecco un'espressione che mostra la sintassi tutto quanto detto sopra:

```
bool casino = rand() % 2;
int bias = 9999
auto whatever = [bias] (int n) mutable -> int {
    if (casino)
        return n * bias;
    else
        return n / bias;
};
```

Domanda 2

La distinzione tra lvalue e rvalue a primo acchitto può essere semplificata con questa spiegazione: un lvalue si trova spesso a destra dell'uguale mentre un rvalue si trova a sinistra. Ma cosa significa veramente questo?

Lvalue sono tutti quei valori che hanno uno spazio dedicato nella memoria dello stack del programma e che quindi possono essere dereferenziati.

Rvalue sono tutti quei valori che sono temporanei (nella riga `a=2;` il 2 è un intero temporaneo che non ha un indirizzo di memoria nello stack ma si trova nello spazio di indirizzamento delle variabili temporanee, che a quanto pare è un mondo a sé); i temporanei non sono dereferenziabili. O almeno così
fino al `C++11`: proprio in questa edizione del nostro linguaggio preferito è stata introdotta la possibilità di ottenere una referenza a rvalue utilizzando l'operatore `&&`. Questo rende possibile appunto ottenere una referenza a valore temporaneo.

A cosa serve una referenza a valore temporaneo? Serve per poter implementare quella che viene chiamata *move semantics*, ovvero un meccanismo per rendere efficiente la creazione di variabili usando invece che un costruttore di copia un costruttore move.

Mentre con il costruttore di copia quando noi istanziamo una certa variabile di classe `A`

```
A a1 = new A(12);
```

`a1` prenderà il valore di `A(12)` facendo una copia dell'istanza temporanea che si crea a destra dell'uguale, con il costruttore di move

```
A a2 = A::move(new A(12));
```

L'istanza temporanea non viene copiata ma viene a tutti gli effetti "spostata" in `a2`, così da evitare il peso di una copia.

Naturalmente questo non ha senso se la classe `A` contiene solo un intero, ma inizia ad acquistare senso se A contiene matrici o oggetti molto pesanti allocati nell'heap: risparmiare la copia di questi è senza dubbio una buona idea.

Il costruttore move viene quindi implementato in modo da rubare tutti i puntatori dell'istanza temporanea, che tanto è temporanea e non serve a nessuno, e darli all'istanza di cui ci importa qualcosa ffettivamente, sotto un esempio:

```
class A
{
private:
    int *n;
public:
    A(A&& _a)
    {
        n = _a.n;
        _a.n = NULL; // si annullano i puntatori del temporaneo
    }
}
```

A questo punto i più curiosi tra di noi si chiederanno perché annullare il puntatore del temporaneo? La risposta è presto detta: non sai mai che fine farà il temporaneo, quindi se vuoi evitare che il caro n venga distrutto dal garbage collector o cose simili devi fare in modo che il suo destino sia separato da quello di `_a`.

Domanda 3

Okay, bisogna affrontare anche questa domanda prima o poi no?

Il template metaprogramming è un metodo di metaprogrammazione che si è venuto a creare per sbaglio (non scherzo, nessuno l'ha voluto, un giorno un tizio ha aperto devcpp e se l'è troato davanti).

Metaprogrammazione è unome altezzoso che diamo a quei programmi che prendono come input altri programmi e anche se sembra una cosa esotica ti assicuro che anche tu caro lettore hai già usato attivamente qualche metaprogramma se hai usato almeno una volta in vita un compilatore.

Nel cpp la metaprogrammazione è resa possibile dal meccanismo dei template, questi infatti ci permettono di definire tipi generici con un certo livello di flessibilità, tanto che usando un po' di ingegno si può andare a definire dei tipi che quando vengono *risolti* a compile time inducono il calcolo ricorsivo o addirittura la specifica di costrutti if.

Presentiamo quindi ora un esempio di implementazione del calcolo dei fattoriali tramite template metaprogramming.

```cpp
#include <iostream>

template<unsigned int n> // caso induttivo
struct factorial {
  enum { value=n*factorial<n-1>::value };
};

template<> // caso base
struct factorial<0>{
  enum { value=1 };
};

int main(){
  // verranno definiti a compile-time i fattoriali da 7 in giù
  cout << factorial<7>::value;
}
```

Presentiamo quindi ora un esempio di implementazione di istruzione condizioale tramite template metaprogramming.

```cpp
#include <iostream>

template <bool condition, class Then, class Else>
struct IF {
  typedef Then RET;
};

// caso particolare del template di sopra
template <class Then, class Else>
```

```
10  struct IF<false, Then, Else> {
11    typedef Else RET;
12  };
13
14  int main(){
15    // definizione diversa a tempo di compilazione in base a sizeof
16    IF< sizeof(int)<sizeof(long), long, int >::RET i;
17    cout << sizeof(i) << endl;
18  }
```

Come si è visto quindi il template metaprogramming è un mezzo che permette di scrivere programmi che eseguono determinate operazioni a compile time invece che a runtime risparmiando tempo di esecuzione. Un altro use case è quello di definire a compile time dei tipi in maniera specifica in base alle caratteristiche della macchina su cui avviene la compilazione.

Domanda 4

Il fatto è che, come per tutte le domande nella vita, la risposta dipende dalla situazione.

Da un lato ci troviamo il vettore che è comodo come un array per quanto riguarda le questioni di indirizzamento all'interno del vettore (letteralmente, io posso fare `vettore[11]` e beccare il 12° elemento del vettore lì che mi aspetta), questo significa che posso trovare un elemento che sta nel mezzo del vettore senza fastidio e con una complessità che rasenta quella di sbucciare un mandarino con una katana (che in realtà secondo me è abbastanza complicato, ma cosa ne so io, purtroppo io ho scelto informatica quella volta... *ndr*).

Da questo punto di vista uno senz'altro direbbe che è molto meglio il vettore della lista: nella lista, se voglio beccare il 12° elemento non ho altro modo se non quello di passare uno a uno i primi 11 elementi.

Ma la lista ha fatto anche lei delle cose buone. Si veda alla voce "inserimento e rimozione di un elemento"; per inserire un nuovo elemnto in una lista abbiamo una complessità $O(1)$, mentre per inserire un mandarino nel vettore abbiamo una complessità ammortizzata costant.

Detto in parole un filo serie il vettore offre accesso a posizioni random con costo costante ma le operazioni di inserimento o rimozione di un elemento possono rivelarsi molto pesanti; conversamente la lista offre inserimento e rimozione con costo costante ma accesso random con costo lineare.

Sono entrambi iterabili ordinatamente con iteratori di tutti i tipi.