Domanda 1

La keyword `virtual` ha principalmente 2 significati, in `C++`.

Il primo è di forzare il late binding dei metodi che la vedono anteposta nella dichiarazione; per late binding intendiamo che, quando abbiamo un metodo di una classe base che viene overridato in una classe figlia e nel main chiamiamo il suddetto metodo da un'istanza (allocata dinamicamente, importante) della classe figlia, il binding del metodo non viene risolto a compile time, bensì a runtime.

```
class Pasta
{
  ...
  virtual void manteca() { ... }
  ...
}

class Amatriciana
  : public Pasta
{
  ...
  void manteca() { ... }
}

int main () {
  Pasta* p = new Amatriciana();
  p.manteca() // se non avessimo dichiarato manteca() come virtual
              // avremmo chiamato Pasta::manteca() invece grazie a
              // virtual chiamiamo Amatriciana::manteca()
}
```

Inoltre, con una precisa sintassi, `virtual` va a indicare che un certo metodo è *puramente virtuale*, ossia che è una firma senza implementazione. La presenza di un metodo puramente virtuale rende la classe stessa puramente virtuale, vale a dire non istanziabile e utilizzabile esclusivamente come classe base da cui derivare altre classi.

```
class Pasticcio
{
  ...
  virtual void sistemaPasticcio() = 0;
  ...
}
```

In ultimo, `virtual` è utilizzata nell'ereditarietà virtuale, un meccanismo che si rende necessario nel momento in cui andiamo a trattare un caso di ereditarietà multipla. Ereditare virtualmente, come nell'esempio qui sotto, previene la doppia chiamata del costruttore della classe base da cui ereditano i due padri della nostra classe figlia.

```
class Persona
{
  ...
}

class Studente
  : virtual public Persona
{
  ...
}

class Faculty
  : virtual public Persona
{
  ...
}

class TA
  : public Faculty
  , public Studente
{
  ...
}
```

Domanda 2

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. Sono organizzati secondo una particolare gerarchia: *Random access* > *Bidirectional* > *Forward* > *Input* | *Output*; inoltre, possono avere l'ulteriore qualificazione di `reverse`, per attraversare il contenitore al contrario, `const`, per prevenire la modifica degli elementi del contenitore durante l'attraversamento, e ovviamente la combinazione di questi ultimi due.

Domanda 3

Un costruttore implicito è un costuttore a un parametro che viene chiamato automaticamente nel caso in cui a una funzione che accetta un'istanza della classe del costruttore si passi un oggetto del tipo accettato dal costruttore stesso; questo processo è detto *conversione implicita di tipo*.

Supponiamo ad esempio di avere una classe `A` simile a

```
class A {
  private:
    int n;
  public:
    A(int _n) {
        n = _n;
    }
};
```

e inoltre una certa funzione `foo()` che riceve un'istanza di `A`

```
void foo(A a) {
    // do something idk
}
```

La conversione implicita di tipo avviene quando chiamiamo la funzione `foo()` passandole un intero come input:

```
int main() {
    foo(A(5));  // tutto a posto
    foo(5);     // OK: conversione implicita di tipo
}
```

È senza dubbio un comportamento peculiare che può risultare comodo in alcune circostanze, ma confusionario e poco desiderabile in altre. Dovesse rendersi necessario, possiamo impedire al compilatore di accettare la conversione implicita anteponendo la keyword `explicit` nella dichiarazione del costruttore incriminato.

```
...
  public:
    explicit A(int _n) { ... };
...
...
int main() {
    foo(A(5));  // tutto a posto, come prima
    foo(5);     // NON COMPILA, abbiamo usato explicit
}
```

Domanda 4

Allora, si può introdurre la metaprogrammazione in `C++` solo se si è dei masochisti perditempo; in tale evenienza il meccansimo del linguaggio che si va a sfrutare per introdurre metaprogrammazione è il templating.

Il templating permette di specificare tipi che vengono risolti solo a compile time, inoltre questo meccanismo è abbastanza flessible da permettere di implementare controllo condizionale e ricorsione.

Io ripropongo qui come esempio di utilizzo del template metaprogramming il calssico calcolo del fattoriale grazie ai template.

```
#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 };
};
```

```
12
13  int main(){
14    // verranno definiti a compile-time i fattoriali da 7 in giù
15    cout << factorial<7>::value;
16  }
```

Il template metaprogramming `C++` in 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.