Lista di cose a cui stare attenti (cose tricky o scherzetti, citando letteralmente il prof)

Occhio nelle modellazioni

Quando si scrivono le classi, attenzione che:

- *se si ha una classe base di una gerarchia, si mette il distruttore virtuale* (o in generale quando la classe base non è istanziabile ma nessuno dei metodi richiesti è virtuale puro)

- *per ogni classe si ricordi di creare un costruttore*, a seconda dei tipi di campi privati. In alcuni esercizi viene esplicitamente richiesto di crearli, buona norma metterli.

- *potrebbe anche crearsi l'overloading derivato dell'operatore di uguaglianza, come:*
  Esempio in corso: caso Capo (super) e Jeans (Derivata)
  In Capo (campi privati appunto brand e taglia, quindi basta mettere campi privati della superclasse)
  ```
  virtual bool operator==(const Capo& c) const{
          return brand == c.brand && taglia == c.taglia;
      }
  ```
  In Jeans (campo privato fondo, quindi basterà mettere campo privato della sottoclasse)
  ```
  virtual bool operator==(const Capo& c) const{
  return typeid(const Jeans&) == typeid(c) && Capo::operator==(c)
  && fondo == (static_cast<const Jeans&>(c)).fondo;
  }
  ```

- *per i campi privati, va sempre creato un metodo const di ritorno del campo*

- *quando accedo ad un iteratore, normalmente (90% dei casi), devo accedere correttamente dereferenziandolo*
  Quindi facendo (*it)-> come:
  ```
  (*it)->get_traffico_ore()
  ```

- nel caso di costruzione di sottooggetti*, si verifichi se è il caso di usare anche il costruttore della classe base per costruire alcuni campi della classe derivata* (può succedere, esempi QabstractButton e similari nelle modellazioni)

- *quando il metodo non fa side effects, si mette sempre il const*

- può capitare che si debba creare *un metodo di clonazione del puntatore*, polimorfo, tra superclasse e sottoclassi
  (per intenderci una cosa tipo:
  ```
  virtual SuperClasse* clone() const =0
  virtual SottoClasse* clone const(return new SottoClasse(*this))
  ```

- negli esercizi che comprendono iteratori, *quando si deve trovare un elemento preciso* (per intenderci, un ciclo dove ho un booleano che ferma il ciclo quando trovo l'elemento), *è necessario sottrarre all'iteratore una posizione per riportarlo al punto corretto*.

- negli esercizi che comprendono iteratori, *quando si cancella un iteratore in una posizione è necessario sottrarre all'iteratore una posizione per riportarlo al punto corretto*.

- nel caso io debba fare una *copia degli elementi di un vettore*, *prima si esegue la push_back* e *poi la delete del puntatore interessato*

- *nel caso si abbia un vettore/lista costante, attenzione a fare i cast giusti*

Esempio:
```
list<const Abbonato*> abbonatiOltreSoglia()
```
Quando voglio l'iteratore di una sottoclasse devo fare anche il const_cast:
```
AbbonatoTempo*a=dynamic_cast<AbbonatoTempo*>(const_cast<Abbonato*>(
*it));
```

Nelle modellazioni di classi "container"

Nell'ordine:

- classe istanziata ad un campo di tipo T generico puntatore e facente parte di template class T
- costruttore di default: inizializza a nullptr il puntatore o i puntatori, se più di uno
  ```
  SmartP(): p(nullptr) {}
  ```
- costruttore di copia profonda: il puntatore alloca uno spazio uguale al proprio parametro e si inizializza. Esempio concreto:
  ```
  SmartP(const SmartP<T>& s):p(new T(*(s.p))){}
  ```
- distruttore profondo: semplice delete del puntatore
- assegnazione profonda:
  1) si controlla che this e riferimento costante siano diversi
  2) si esegue la delete del puntatore
  3) si crea nello heap un puntatore di grandezza pari alla dimensione del puntatore del rif. costante
  Esempio pratico:
  ```
  SmartP<T>& operator=(const SmartP<T>& s){
      if(this!=&s)
      {
          delete p;
          p = new T(*s.p);

      }
      return *this;
  }
  ```
- possibile overloading dell'operatore di chiamata di funzione, ad esempio:
  ```
  SmartP<T> operator ()(T*& p){return SmartP(p);}
  ```
- operatori di dereferenziazione e selezione di membro ridefiniti:

```
   T& operator*()const{
    return *p;
   }
//operatore di selezione di membro
 T* operator->()const
{return p;}
```

Possono essere presenti delle classi iteratore o classi che scorrono elementi e:

- possiederà un campo puntatore privato della classe container (esempio, la classe dList che ha la classe Nodo come classe privata, due campi che sono first e last e dentro la classe iterator esiste il puntatore alla classe Nodo*
- costruttore di default che mette a nullptr il puntatore
- operatore di incremento pre/postfisso, fatti come si vede sotto (decremento è totalmente speculare, solo che nel caso dei puntatori si deve avere una copia/puntatore dell'elemento precedente)

Si ricordi che l'operatore prefisso, prima incrementa poi va avanti nel puntatore mentre il postfisso prima copia il campo this e poi incrementa. Il postfisso prende come argomento un int fittizio.

```
// Define prefix increment operator.
Point& Point::operator++()
{
   _x++;
   _y++;
   return *this;
}

// Define postfix increment operator.
Point Point::operator++(int)
{
   Point temp = *this;
   ++*this;
   return temp;
}
```

- overloading degli operatori di ==/!= (uguale/diverso), semplicemente mettendo diversi i campi puntati.
  Ad esempio:
  ```
  bool operator==(const const_iterator & cit) const {
   return ptr == cit.ptr;
  }
  bool operator!=(const const_iterator& cit) const {
   return ptr != cit.ptr;
  }
  ```
- creazioni dei metodi begin e end sapendo che puntano o al primo/ultimo elemento e semplicemente:

```
const_iterator begin() const {
    return first; // nodo* => const_iterator
  }

  const_iterator end() const {
    if(first == nullptr) return const_iterator(nullptr,false);
    return const_iterator(last+1,true);
  }
```

Nel caso invece di un vettore con una dimensione (size) si ricordi di inizializzare in questo modo nella lista di inizializzazione (riporto per comodità costruttore di default, costruttore di copia e operatore di assegnazione):

```
Vettore(const T& x=T()): a(size==0 ? nullptr : new T[sz]){
        for(int i=0; i<size; i++) a[i]=x;
    }
    Vettore(const Vettore& v): a(size == 0 ? nullptr : new T[sz]{
        for(int i=0; i<size; i++) a[i]=v.a[i];
    }
    Vettore& operator=(const Vettore& v){
        if(this != &v){
            delete[] a;
            a = sz == 0 ? nullptr : new T[sz];
            for(int i=0; i<sz; ++i) a[i]=v.a[i];
        }
        return *this;
    }
```

Occhio ai const

A*p3 = new C();
p3->k();

Il programma utile è:

```
# include <iostream>
using namespace std;

   class A {
   protected:
   virtual void j() { cout<<" A::j "; }
   public:
   virtual void g() const { cout <<" A::g "; }
   virtual void f() { cout <<" A::f "; g(); j(); }
   void m() { cout <<" A::m "; g(); j(); }
   virtual void k() { cout <<" A::k "; j(); m(); }
   virtual A* n() { cout <<" A::n "; return this; }
   };

   class C: public A {
   private:
   void j() { cout <<" C::j "; }
   public:
   virtual void g() { cout <<" C::g "; }
   void m() { cout <<" C::m "; g(); j(); }
   void k() const { cout <<" C::k "; k(); }
   };
```

Lui stampa perché il c::g() non si ha il const, per evitare i soliti side effects.

```
   A::k  C::j  A::m  A::g  C::j
```

Se metto in c::g() il const, ecco che lui stampa:

```
   A::k  C::j  A::m  C::g  C::j
```

Attenzione: i const sono considerate a tutti gli effetti come funzioni con tipo diverso, pertanto cambia molto tra una funzione const e una funzione non const. Inoltre, le funzioni const possono invocare solo funzioni costanti

Occhio ai tipi covarianti

Si può avere questa casistica quando:

```
   class A {
   protected:
   virtual void j() { cout<<" A::j "; }
   public:
   virtual void g() const { cout <<" A::g "; }
   virtual void f() { cout <<" A::f "; g(); j(); }
   void m() { cout <<" A::m "; g(); j(); }
   virtual void k() { cout <<" A::k "; j(); m(); }
   virtual A* n() { cout <<" A::n "; return this; }
   };
```

```
class B: public A {
public:
virtual void g() const override { cout <<" B::g "; }
virtual void m() { cout <<" B::m "; g(); j(); }
void k() { cout <<" B::k "; A::n(); }
A* n() override { cout <<" B::n "; return this; }
};

class D: public B {
protected:
void j() { cout <<" D::j "; }
public:
B* n() final { cout <<" D::n "; return this; }
void m() { cout <<" D::m "; g(); j(); }
};
```

Ho un'invocazione:

```
(p4->n())->m();
```

che genera le stampe: `D::n  A::m  B::g  D::j`

In questo caso specifico, si nota che viene tornato un *this* che viene dichiarato virtual in B e si va a chiamarlo in D; è un tipo covariante in quanto contemporaneamente è un puntatore di tipo D ed un A e si decide dove andare a seconda del binding di tipo. Si nota anche l'uso del final per dichiarare la fine delle dichiarazioni virtuali.

Occhio agli errori a runtime

Per esempio, considerando:

```
A* p2 = new B();
(static_cast<C*>(p2))->k();
```

Il cast funziona correttamente, ed invoca c::k(), ma appunto si ha una chiamata infinita del metodo k().

```
void k() const { cout <<" C::k "; k(); }
```

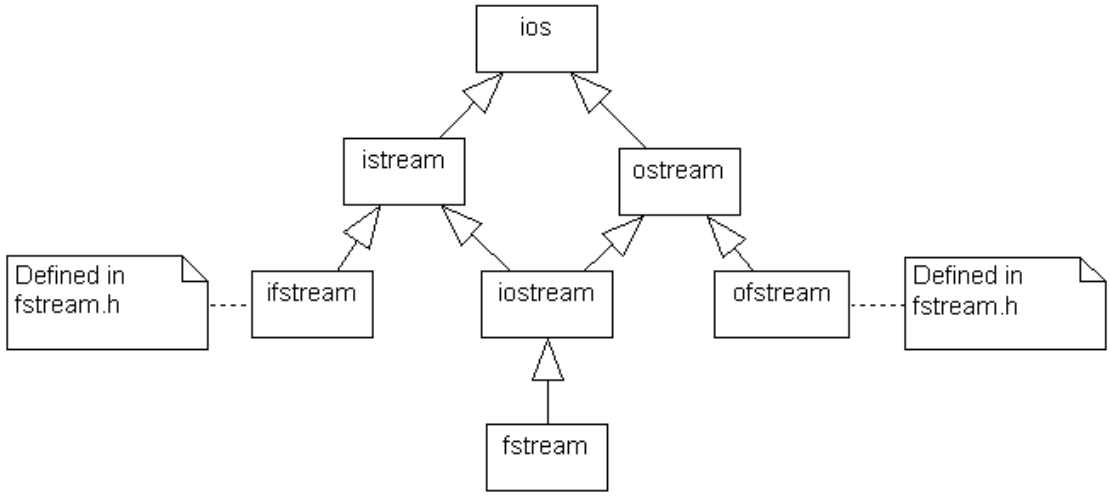
Occhio al dynamic_cast

- Usi **illeciti**:
  - conversione di un puntatore a *genitore* in un puntatore a *figlio*, quando, nonostante il genitore abbia almeno una funzione virtuale, il puntatore è inizializzato a puntare un oggetto genitore: in questo caso il compilatore NON SI LAMENTA, ma all'esecuzione si genera un puntatore nullo.

```
class Mamma {virtual void funz( ){ }};
class Figlia : public Mamma { };

int main( )
{
Mamma * m = new Mamma;
Figlia * f;
f = dynamic_cast<Figlia *>(m);
if(!f) return 77;
}
```

Occhio alla gerarchia iostream



Sapendo per esempio che:

- cin è parte di istream
- cout è parte di ostream
- cerr è parte di fstream

Per esempio il seguente esercizio:

```
#include<iostream>
#include<fstream>
#include<typeinfo>
using namespace std;

class C { public: virtual ~C() {} };

template <typename T1, typename T2>
bool Fun(T1* t1, T2& t2) {
    if(typeid(t1) == typeid(C*)) return false;
    return typeid(t1) == typeid(&t2);
}

int main() {
    ifstream f("pippo");
    fstream g("pluto"), h("zagor");
    iostream* p = &h;
    C c1,c2;
    cout << Fun(&cout,cin) << endl; // stampa: 0
    cout << Fun(&cout,cerr) << endl; // stampa: 1
    cout << Fun(p,h) << endl; // stampa: 0
    cout << Fun(&f,*p) << endl; // stampa: 0
    cout << Fun(&g,h) << endl; // stampa: 1
    cout << Fun(&c1,c2) << endl; // stampa: 0
}
```

In questo caso si vede in particolare dalla penultima stampa che stampa 1 quando tipo statico/dinamico coincidono e la stampa dell'ultima riga determina la prima condizione. Basti controllare le relazioni di gerarchia ed ereditarietà ed è più affrontabile di quanto sembri.

Occhio agli esercizi di ridefinizione dei costruttori di copia/operatori di assegnazione con comportamento simile a quelli standard

Consideriamo una situazione di questo tipo:

```
class B {
private:
      list<double>* ptr;
      virtual void m() =0;
};

class C: virtual public B {};

class D: virtual public B {
private:
      int x;
};

class E: public C, public D {
private:
      vector<int*> v;
public:
      void m() {}

      // ridefinizione del costruttore di assegnazione di E
      E(const E& e){

      }
```

In questo caso specifico, si devono costruire tutti gli oggetti in ordine di derivazione, partendo dalla base, nella semplice lista di assegnazione. Essendo che la base per C e D risulta essere virtuale, si costruisce una sola volta un particolare oggetto e si risolve facilmente.

```
E(const E& e): B(e), C(e), D(e), v(e.v) {}
```

Per l'assegnazione vediamo:

```
#include <iostream>

using namespace std;

class Z { private: int x; };

class B { private: Z x; };

class D: public B {
      private:
      Z y;
      public:
      // ridefinizione di operator=
      D& operator=(const D&);
};

D& D::operator= (const D& x){
      this->B::operator=(x);
      //mi devo occupare del campo dati
```

```
        y=x.y;
        return *this;
}

int main(){

}
```

L'idea è di usare quello che viene fornito direttamente dalle basi dirette, a seconda della relativa posizione nella gerarchia e costruire poi il proprio sotto-oggetto, in merito naturalmente al riferimento costante e lo *this. Naturalmente si aspira ad un comportamento simile all'avere: x.y=y;

Occhio allo scoping

Nota di contorno, lo scoping blocca il late binding; senza usare termini fighi, significa semplicemente che vado a chiamare effettivamente la funzione indicata, qualunque sia la possibile funzione ridefinita. Quindi:

- se io avessi una funzione f() dichiara virtuale in una classe base A e viene fatto l'overloading (virtuale, stessa segnatura) del metodo f(), ma faccio A::f() andrò ad invocare quella funzione per certo

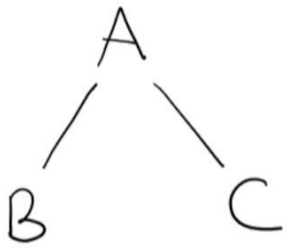
Occhio alla chiamata di metodi con classi sullo stesso livello della gerarchia

```cpp
class A {
    public:
    virtual ~A() {};
    virtual void m() {std::cout<<"A::m";};
};

class B: public A {
    public:
    void m() {std::cout<<"B::m";};
};

class C: public A {
    public:
    void m() {std::cout<<"C::m";};
};

int main() {
    A* pa = new A(); A* pb = new B(); A* pc = new C();
    B* b  = static_cast<B*>(pc);
    b->m();
}
```



Mi riferisco a una giusta domanda sul gruppo Telegram, in cui si chiede la validità di questo cast. La soddisfacente risposta è la seguente:

È valido. Il static cast l'unico controllo che fa è a compile time e verifica che l'argomento statico sia dello stesso ramo dell'argomento di destinazione . In questo caso pc è un A. Fa parte dello stesso ramo di B* quindi il cast è lecito. L'espressione dello static cast è un B* che punta ad un C. Richiamare m() su questa espressione funziona e trova dalla virtual table B::m(). (Se B::m() non fosse virtuale sarebbe ERT). Salta subito nel tipo dinamico C e trova C::m() virtuale e compatibile . Funziona solo perché trova nel tipo dinamico un virtuale compatibile. Se non ci fosse C::m() e neanche A::m() ereditato. oppure ci fosse ma non più in overriding sarebbe errore runtime . Stampa C::m()*

Occhio al tipo di ritorno e azione sui vector (se dereferenziati o meno)

Se io avessi una cosa tipo:
`std::vector<HwEngineer> fire(double s)`
Quando devo eseguire un'operazione diretta su un certo puntatore, bisogna ricordarsi dello *, in particolare:
`if(q) v.push_back(*q);`

Se invece si ha una situazione del tipo:
`std::vector<SwEngineer*> masterInf()`
Ecco che le cose cambiano leggermente, ma cambiano.
`if(q) v.push_back(q);`

Appunti e note di contorno

- Normalmente, è bene mettere protected i campi privati in caso di ereditarietà di sottoclassi, in modo tale che alle stesse tutto sia disponibile in un certo momento.
- Occhio a scrivere bene i campi statici: essi vanno dichiarati pubblici nella classe di riferimento, poi all'esterno della stessa vanno ridichiarati scrivendo tipo di riferimento (class)::campo=(valore)