Definisci la classe Fallimento per lanciare eccezione

Soluzione:

```
// Modo compatto

class Fallimento{};

        throw Fallimento();

// Modo esteso

class Fallimento{
        private:
                std::string msg;
        public:
                Fallimento(std::string m): msg(m) {}
};

// Altrimenti

Lanciamo std::string("Null");
```

CASI CON * o &

- list/vector<Tipo*>
  - p = it→metodo() oppure vector.push_back(it);

- list/vector<Tipo>
  (*it).metodo() oppure vector.push_back(*it);

OVERLOADING ...

```
T operator* () const {
    if (p == nullptr) throw std::exception();
    return *p;
}

T* operator→ () const {
    if (p == nullptr) throw std::exception();
    return p;
}
```

* → ALIAS

→ OGGETTO PUNTATO

METODI

- INSERIMENTO => PUSH_BACK
- CANCELLAZIONE => ERASE
- DEALLOCAZIONE => DELETE

- Cancellazione dell'oggetto = erase
- Dealloca l'oggetto e poi elimina = delete *it / it = lst.erase(it);

Should we delete before or after erase for an pointer in the vector?

"itr" must be used like this;

```
for (vector<foo*>::iterator itr = bar.begin(); itr != bar.end(); )
{
   delete (*itr);
   itr = bar.erase(itr);
}
```

La risposta giusta è: dobbiamo sempre fare delete PRIMA di erase. Il motivo è fondamentale:

1. Quando facciamo erase di un elemento da un vector, l'iteratore/indice a quell'elemento diventa invalido
2. Se facessimo erase prima di delete, staremmo tentando di dereferenziare un puntatore potenzialmente invalido, causando undefined behavior

1. delete deallocca la memoria puntata
2. erase rimuove il puntatore dal container
3. L'ordine è critico perché dopo erase non possiamo più accedere validamente all'elemento

La soluzione più elegante e moderna sarebbe usare smart pointer (std::unique_ptr) che gestiscono automaticamente la deallocazione quando l'elemento viene rimosso dal container.

Esercizio 3

Si considerino le seguenti dichiarazioni di classi di qualche libreria grafica, dove gli oggetti delle classi `Container`, `Component`, `Button` e `MenuItem` sono chiamati, rispettivamente, contenitori, componenti, pulsanti ed entrate di menu.

```
class Component;

class Container {
public:
  virtual ~Container();
  vector<Component*> getComponents() const;
};

class Component: public Container {};

class Button: public Component {
public:
  vector<Container*> getContainers() const;
};

class MenuItem: public Button {
public:
  void setEnabled(bool b = true);
};

class NoButton {};
```

Si assumino i seguenti fatti.

1. Il comportamento del metodo `getComponents()` della classe `Container` è il seguente: `c.getComponents()` ritorna un vector di puntatori a tutte le componenti inserite nel contenitore `c`; se `c` non ha alcuna componente allora ritorna un vector vuoto.
2. Il comportamento del metodo `getContainers()` della classe `Button` è il seguente: `b.getContainers()` ritorna un vector di puntatori a tutti i contenitori che contengono il pulsante `b`; se `b` non appartiene ad alcun contenitore allora ritorna un vector vuoto.

3. Il comportamento del metodo `setEnabled()` della classe `MenuItem` è il seguente: `mi.setEnabled(b)` abilita (con `b==true`) o disabilita (con `b==false`) l'entrata di menu `mi`.

Definire una funzione `Button** Fun(const Container&)` con il seguente comportamento: in ogni invocazione `Fun(c)`

1. Se `c` contiene almeno una componente `Button` allora

   ritorna un puntatore alla prima cella di un array dinamico di puntatori a pulsanti contenente tutti e soli i puntatori ai pulsanti che sono componenti del contenitore `c`; inoltre tutte le componenti del contenitore `c` che sono una entrata di menu e sono contenute in almeno 2 contenitori devono essere disabilitate.

2. Se invece `c` non contiene nessuna componente `Button` allora ritorna il puntatore nullo.

```
Button **Fun(const Container &c){

        auto v = c.getComponents();
        // tipo → vector<Button*>

        for(vector<Button*>::iterator it = v.begin(); it ≠ v.end(); ++it){
                Button* b = dynamic_cast<Button*>(*it);
                if(b){
                        v.push_back(b);
                        MenuItem *m = dynamic_cast<MenuItem*>(*b);
                        if(m && m→getContainers().size() > 2)
                                m→setEnabled(false);

                        typeid/std::bad_cast → #include <typeinfo>
                }
        }

        return **v;
        // return &v[0];

}
```

```cpp
Button** Fun(const Container& c) {
    vector<button*> aux;
    vector<Container*> cont;
    for(auto it = cont.begin(); it != cont.end(); ++it){
        Button* b = dynamic_cast<Button*>(*it);
        if(b){
            aux.push_back(b);
            MenuItem* m = dynamic_cast<MenuItem*>(*it);
            if(m && m->getContainers().size() > 1)
                m->setEnabled(false);
        }
    }
    if(aux.empty()) return nullptr;
    return &aux[0];
}

Button** Fun(const Container& c) {
    vector<Button*> v;
    for(auto* el : c.getComponents()){
        auto* b = dynamic_cast<Button*>(el);
        if(b){
            v.push_back(b);
            auto* m = dynamic_cast<MenuItem*>(el);
            if(m && m->getContainers().size()>1){
                m->setEnabled(false);
            }
        }
    }
    if(v.empty()) return nullptr;
    return &v[0];
}
```

Quesito 3:

quesito3.cpp 1

```
template <class A,class B>

A*Fun(A*p){return dynamic_cast<B*>(p);}

main(){
    C c; fun<A,B>(&c);
    if(fun<A,B>(new C())==0) cout<<"Alan";
    if(dynamic_cast<C*>(new B())==0) cout<<"Turing";
    A*p=fun<D,B>(new D());
}
```

	Vero	Falso	Possibile
A<=B		X	
A<=C		X	
A<=D		X	
B<=A	X		
B<=C		X	
B<=D			

	Vero	Falso	Possibile
C<=A	X		
C<=B		X	
C<=D		X	
D<=A	X		
D<=B	X		
D<=C			X

```
class A {
public:
  virtual void m() =0;
};

class B: virtual public A {};

class C: virtual public A {
public:
  virtual void m() {}
};

class D: public B, public C {
public:
  virtual void m() {}
};

class E: public D {};

class F: public E {};

char G(A* p, B& r) {
  C* pc = dynamic_cast<E*>(&r);
  if(pc && typeid(*p)==typeid(r)) return 'G';
  if(!dynamic_cast<E*>(&r) && dynamic_cast<D*>(p)) return 'Z';
  if(!dynamic_cast<F*>(pc)) return 'A';
  else if(typeid(*p)==typeid(E)) return 'S';
  return 'E';
}
```

Si consideri inoltre il seguente statement.

```
  cout << G(new X1,*new Y1) << G(new X2,*new Y2) << G(new X3,*new Y3) << G(new X4,*new Y4)
       << G(new X5,*new Y5) << G(new X6,*new Y6) << G(new X7,*new Y7) << G(new X8,*new Y8);
```

Definire opportunamente le incognite di tipo `Xi` e `Yi` tra i tipi `A`, `B`, `C`, `D`, `E`, `F` della precedente gerarchia in modo tale che:

1. Lo statement non includa piú di una chiamata della funzione `G` con gli stessi parametri attuali
2. La compilazione dello statement non produca illegalità
3. L'esecuzione dello statement non provochi errori a run-time
4. L'esecuzione dello statement produca in output esattamente la stampa **SAGGEZZA**.