```
/*

Esercizio 1

*/

/*

Chiamata A: (1,2) - (1) - (1)
Chiamata B: (10) - (10) - (10)
Chiamata C: (5,6) - (6) - (6) -> ADL
Chiamata D: (5,6) - (5) - (5)
Chiamata E: / - / - /
Chiamata F: (7,8) - (7,8) - (8)
Chiamata G: (7,8) - (7,8) - (7)
Chiamata H: (9) - (9) - (9) -> ADL
Chiamata I: (11,12) - (11) - (11)
Chiamata J: (3,4) - / - / -> ADL

*/



/*

Esercizio 2

*/

=== 1 ===
B::f(int)
B::g(int)
B::f(int)
C::g(int)
=== 2 ===
A::f(double)
B::f(int)
C::g(int)
=== 3 ===
B::f(int)
C::g(int)
=== 4 ===
Destructor C::~C()
Destructor B::~B()
Destructor A::~A()
Destructor B::~B()
Destructor A::~A()


/*

Esercizio 3

*/

/*

Possiamo notare come in questo codice il principio SOLID che viene violato è ISP.
Interface Segregation Principle dice infatti che bisogna cercare, per quanto possibile, di separare tutte le interfacce che possono essere usate separatamente.
Questo porta vantaggi come la possibilità di modificare parti di codice senza intaccare le altre già funzionanti e evita la propagazione di errori.
Un possibile modo per farlo in questo caso potrebbe essere il seguente.

*/

class Var : public Expr {
  void print(name()) const;

  double eval(const Var_Bindings& vb) const {
    return vb[name()];
  }
};
```

```
72
73  class Const : public Expr {
74    void print(value()) const;
75
76    double eval(const Var_Bindings& vb) const {
77      return value();
78    }
79  };
80
81  class Add : public Expr {
82    void print(arg1, arg2) const {
83      arg1.print();
84      print(" + ");
85      arg2.print();
86    }
87
88    double eval(const Var_Bindings& vb) const {
89      return arg1().eval(vb) + arg2().eval(vb);
90    }
91  };
92
93  class Sub : public Expr {
94    void print(arg1, arg2) const {
95      arg1.print();
96      print(" - ");
97      arg2.print();
98    }
99
100   double eval(const Var_Bindings& vb) const {
101     return arg1().eval(vb) - arg2().eval(vb);
102   }
103 };
104
105
106 class Expr {
107 public:
108   Kind kind;
109   // ...
110
111   // Overriding
112   virtual void print() const = 0;
113   virtual double eval(cons Var_Bindings& vb) const = 0;
114 };
115
116 /*
117
118 Esercizio 4
119
120 */
121
122 #include <string>
123 #include <vector>
124 #include <iostream>
125
126 typedef std::vector<std::string> vect;
127 typedef std::vector<std::string>::iterator iter;
128
129 void f(const vect& v) {
130   iter i = std::find(v.begin(), v.end(), "inizio");
131   iter i_end = std::find(v.begin(), v.end(), "fine");
132   while (i != i_end) {
133       std::cout << *i << std::endl;
134     ++i;
135   }
136
137
138 void g(vect& v) {
139   iter i = v.begin(), i_end = v.end();
140
141   // Inserisco un controllo (come nella funzione 'f') in caso il vettore fosse vuoto.
142   // Gestiamo quindi il caso in cui i=i_end.
143   v.insert(++i, "prima");
144   v.insert(++i, "dopo");
```

```
145
146     i=v.begin();
147     while (i != v.end()) {
148       std::cout << *i << std::endl;
149       ++i;
150     }
151   }
152
153   /*
154
155   Esercizio 5
156
157   */
158
159   #include <iostream>
160   #include <list>
161   #include <iterator>
162
163   template<typename Iter1, typename Iter2, typename Out, typename Pred>
164   Out tranform
165   (Iter1 i1_first, Iter1 i1_last, Iter2 i2_first, Iter2 i2_last, Out i3_first, Pred p) {
166
167     for( ; i1_first != i2_last; i1_first++, i2_first++, i3_first++) {
168       *i3_first = p(*i1_first, *i2_first);
169     }
170
171     return i3_first;
172   }
173
174   double media(int arg1, int arg2) {
175     double m = arg1 + arg2;
176     m = m/2;
177
178     return m;
179   }
180
181   int main() {
182     std::list<int> list1 = {1, 5, 7, 9, 6};
183     std::list<int> list2 = {3, 6, 2, 9, 5};
184
185     // Iteratore di output in cui inserire i valori delle medie calcolate.
186     // Verranno stampati su standard di output separati da una tabulazione.
187     std::ostream_iterator<double> out (std::cout, "\t");
188
189     transform(list1.begin(), list1.end(), list2.begin(), list2.end(), out.begin(),
       media);
190
191     return 0;
192   }
193
194
195   /*
196
197   Esercizio 6
198
199   */
200
201   /*
202
203   All'interno della funzione 'foo' quando viene utilizzata la funzione 'job' notiamo
       che 'pc' è stato già allocato, questo vuol dire che se l'allocazione di 'C' (che
       viene poi passato a 'pc2') non va a buon fine 'pc' non verrà eliminato.
204   Per questo è necessario utilizzare un try-catch nella parte di codice sensibile
       oppure implementare l'idioma RAII.
205
206   */
207
208   // Gestione con try-catch
209
210   struct C { ~C() {} /* ... */ //};
211   struct D : public C { /* ... */ };
212   void job(const C* pc1, const C* pc2){}
213
```

```
214   void foo() {
215     C* pc = new D();
216
217     try{
218       C* pk = new C();
219       try {
220         job(pc, pk);
221       }
222       catch(...) {
223         delete pk;
224         throw;
225       }
226       delete pk;
227       delete pc;
228     }
229     catch(...) {
230       delete pc;
231       throw;
232     }
233   }
234
235   // Gestione RAII e smart pointer
236
237   struct C { ~C() {} /* ... */ //};
238   struct D : public C { /* ... */ };
239   void job(const C* pc1, const C* pc2){}
240
241   void foo() {
242     std::unique_ptr<D> pc (new D());
243     std::unique_ptr<C> pk (new C());
244
245     job(pc.get(), pk.get();
246   }
247
```