

**chiamata A**: Nessun valore è definito dall'utente, i tipi di dato sono built int, dunque non si applica l’argument dependent lookup. Siamo nel namespace globale.

Candidata: 3  
Utilizzabile: 3, per il primo argomento converto int -> double (conversione std)  
Migliore: 3

**chiamata B:** Il nome della funzione è qualificato, devo guardare alla foo che si trova nel namespace N.  
Inizio la ricerca a partire dal namespace N. (Se ci fossero stati tipi definiti dall’utente non si sarebbe comunque applicato l’argument dependent lookup, che si applica solo alle chiamate non qualificate)

Candidate: 1, 2. La 3 non è visibile.  
Utilizzabili: 1, 2  
Migliore: nella #1 abbiamo 2 match perfetti, nella #2 abbiamo 2 conversioni std. La migliore è la 1.

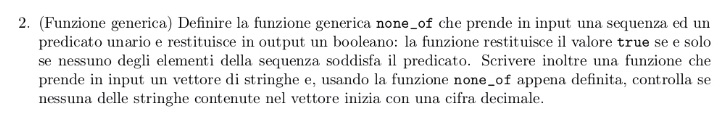
**chiamata C:** prima di effettuare la chiamata C inserisco la direttiva di using (è possibile aprire al bisogno il namespace N).

Candidata: 3, N non viene aperto  
Migliore: 3

**chiamata D:** using N::foo è una dichiarazione di using, dunque devo prendere in N tutte le dichiarazioi di fooe faccio come se fossero dichiarate nel punto in cui ho la dichiaraizione using

Candidate: 3, 1, 2  
Utilizzabili: 1, 2, 3

Nella #2 abbiamo 2 conversioni standard, nella #3 invece abbiamo 1 conversione standard e un match perfetto  
Migliore: 1



template <typename InIter, typename UnaryPred>

bool

none\_of(InIter first, InIter last, UnaryPred pred){

for ( ; first != last; ++first) { // gestisce anche il caso di sequenza vuota

if (pred(\*first)) // se è vero trovo l'elemento che soddisfa il predicato

return false;

} // ho scorso la sequenza e non ho trovato nulla che soddisfà il predicato

return true;

}

void foo(const std::vector<std::string>& vs) {

bool nessuna = none\_of(vs.begin(), vs.end(), inizia\_con\_cifra);

}

Date le operazioni utilizzo un input iterator

bool inizia\_con\_cifra(const std::stringa& s){

if (s.empty()) // stringa vuota

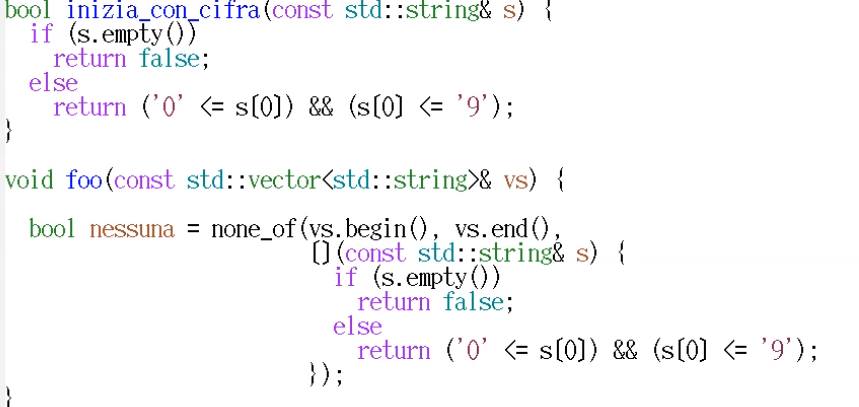
return false;

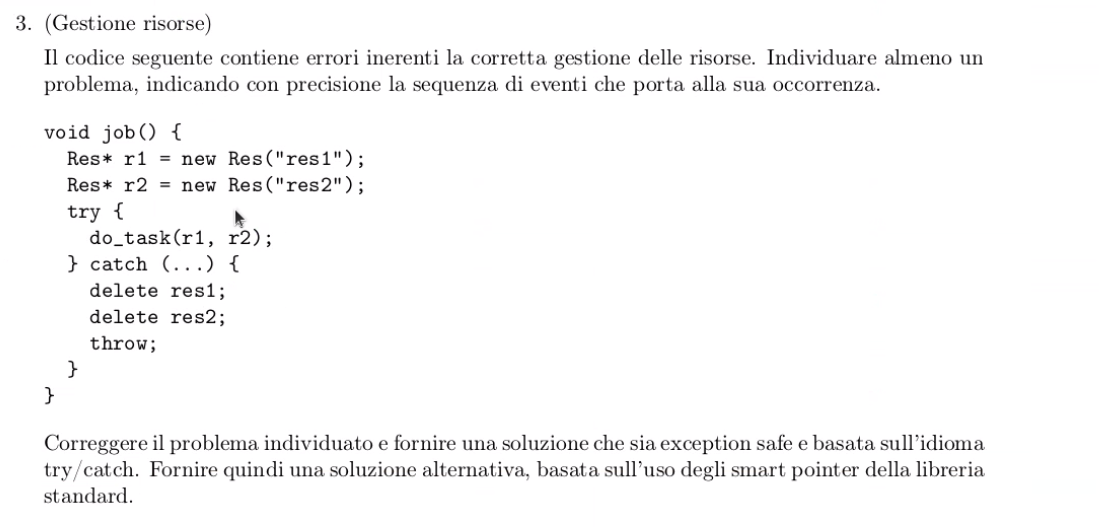
else

return ('0' <= s[0]) && (s[0] <= '9');

}

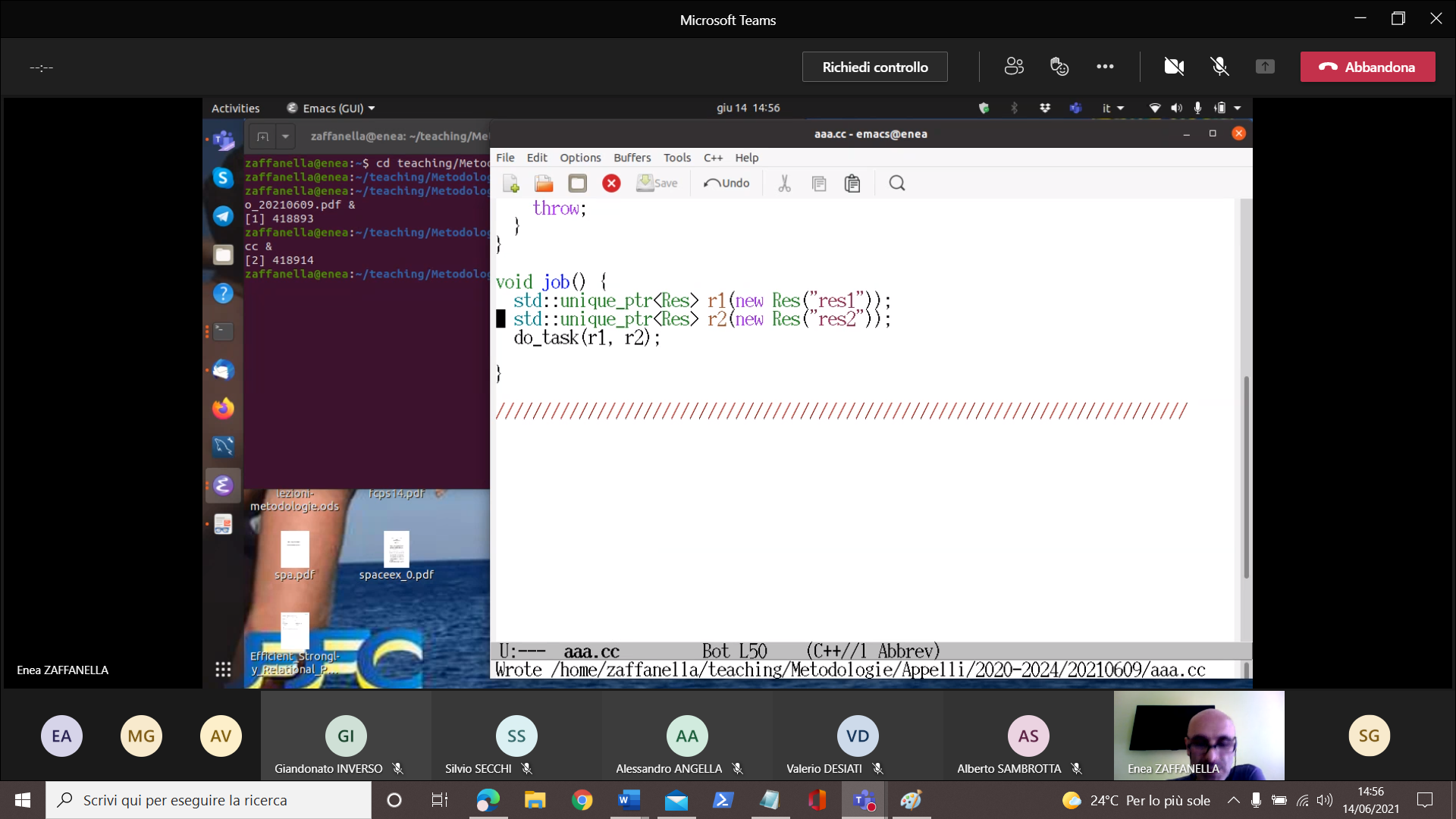
Oppure, utilizzando la lambda expression:



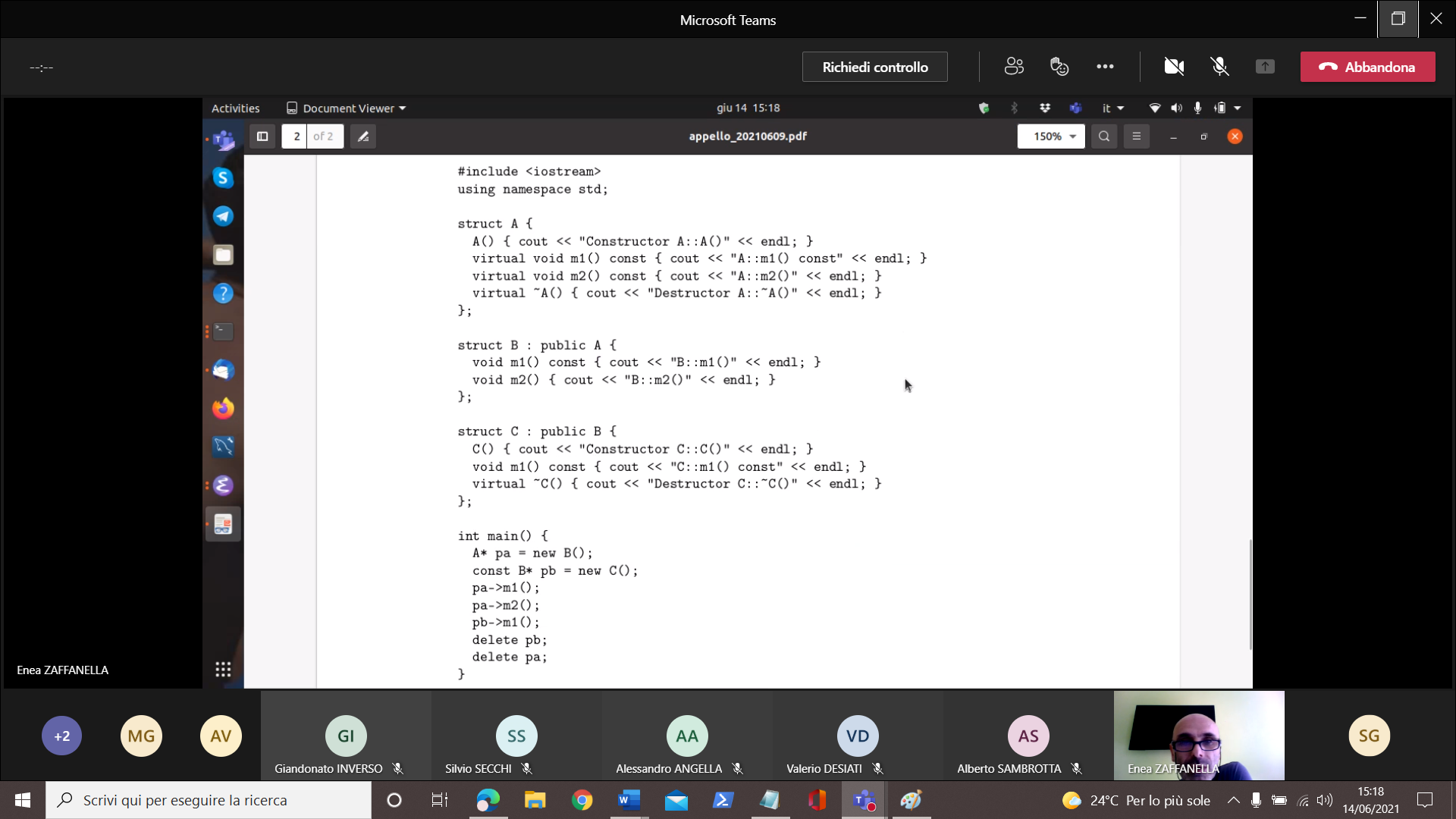


Bisogna essere exception safe, anche in presenza di eccezioni devo comportarmi correttamente circa l'utilizzo delle risorse.  
In questo codice rilascio le risorse solo se le cose vanno male, causando memory leak.  
Violo il single responsability principle: ogni blocco try deve gesitire una sola risorsa alla volta.





Indicare l’output del programma:



Osserviamo: abbiamo la struct A che funzione da classe base. In A ci sono 3 metodi virtuali e il costruttore di default.   
Nella struct B ci sono i metodi m1 e m2 non marcati virtual ma che potrebbero esserlo: B::m1 fa l’overriding del metodo m1 della classe base A. B::m2 invece non fa l’overriding e di conseguenza non è virtuale (questo perché in A::m2 abbiamo il qualificatore const, in B::m2 no).  
Nella struct B c’è anche il distruttore virtuale (poiché anche la classe base è virtuale).  
Nella struct C c’è il metodo m1() che fa override di m1 della classe B che a sua volta fa override della funzione m1() della classe A.

--- MAIN ---

**A\* pa = new B():** costruiamo un oggetto di tipo B(), chiamiamo il costruttore della classe B.  
Il costruttore di default di B costruisce la classe base A, stampa: *costruisco A::A(),* viene invocata poi la parte specifica del costruttore della classe B che non fa nulla.

Il puntatore a B ottenuto con la new viene utilizzato per inizializzare pa, che è un puntatore ad A.  
Parliamo di up cast, dunque possiamo farlo implicitamente (cosa ?) 51:10  
  
**const B\* pb = new C():** viene invocato il costruttore di C, che costruisce prima la classe B, che costruisce prima A: costruisco A, costruisco B, costruisco C in quest’ordine.

Otteniamo un puntatore ad un oggetto C modificabile che utilizzo per inizializzarre un puntatore ad un oggetto B\* non modificabile.

**pa->m1()**: cerco m1 in A (usando il tipo statico), lo trovo. Il compilatore vede che si tratta di è un metodo virtuale invocato tramite un puntatore, l’invocazione non è qualificata e il tipo statico e dinamico potrebbero essere diversi. Quindi il supporto a tempo di esecuzione vede che è stato invocato il metodo m1() della classe A che è virtuale, ma il tipo conreto di pa è un B. Guardo se nella catena di derivazione da A fino B qualcuno ha fato l’overriding di m1, stampo B::m1

**pa->m2():** il supporto a tempo di esecuzione si chiede: nella catena di derivazioni da A a B qualcuno ha fatto l’overriding del metodo m2 della classe A? No, B::m2() non fa l’overrding. Stampo A::m2.

**pb->m1():** Il compilatore cerca m1 nella classe B (staticamente), la trova (nota il qualificatore const), m1 è virtuale.   
Il supporto a tempo di esecuzione si accorge che pb punta ad un oggetto di tipo C, ci si chiede “nella catena di derivazione da B a C c’è qualcuno ha fatto l’overrdiing di m1?”. Sì, l’ha fatto la classe C e stampo C::m1

**delete pb:** invoca il distruttore sull’oggetto puntato. Prima invoco il distruttore su pb (che è un puntatore alla classe B) dunque invoco il distruttore in B che è virtuale (tipo statico e dinamico potrebbero non coincidere). Il lavoro viene demandato al supporto a tempo di esecuzione, si chiede “Il puntatore pb ad un oggetto di tipo D() in realtà sta puntando ad un oggetto di tipo concreto?” Sì, invoco il distruttore di C e distruggo C::C(), poi invoco il distruttore di B, e infine il distruttore di A.

**delete pa:** il supporto a tempo di esecuzione vorrebbe invocare il distruttore di A che è virtuale. Devo vedere se ne è stato fatto l’overriding, l’ha fatto B e dunque invoco il distruttore di B che non stampa nulla e chiama il distruttore di A.

costruisco A::A()  
costruisco B::B() // in realtà non lo stampa

costruisco A::A()  
costruisco B::B()  
costruisco C::C()

B::m1()  
A::m2()  
C::m1()

distruggo C::C()  
distruggo B::B()  
distruggo A::A()

distruggo B::B()  
distruggo A::A()

Nota: ho scritto anche le stampe riguardo al costruttore/distruttore di B nonostante non ci fossero le relative istruzioni di stampa, l’ho fatto solo per tenerne traccia