Funzione anonima locale

`[capture list] (lista parametri) ->return-type { corpo }`

`(lista parametri)` e `->return-type` sono opzionali

```
[] ->int {return 3*3;}          // lista vuota di parametri
[](int x, int y) {return x+y;} // tipo di ritorno implicito: int
[](int x, int y) ->int {return x+y;} // tipo di ritorno esplicito: int
[](int& x) {++x;}               // tipo di ritorno implicito: void
[](int& x) ->void {++x;}        // tipo di ritorno esplicito: void
```

Esercizio Cosa Stampa

```
template<class Functor>
vector<int> find_template(const vector<int>& v, Functor t) {
    vector<int> r;
    for(auto it = v.begin(); it != v.end(); ++it) if (t(*it)) r.push_back(*it);
    return r;
}

unsigned int find_1(const vector<int>& v,int k) {
  vector<int> w = find_template(v, [v,k] (int n) { return n>k; } );
  return w.size();
}

vector<int> find_2(const vector<int>& v) {
  return find_template(v, [v] (int n) { return n<v.size(); } );
}

vector<int> v1 = {3,6,4,6,2,5,-2,4,2}; vector<int> v2 = {-2,-6,4,4,2,5,0,4,2,3,2,0};
```

Prime due chiamate:

find_1(v1, 2) → Prendi tutti i valori (trasformati a Funtore → tipo) > 2 (positivi = unsigned int) dentro a v1 → 6
find_2(v2, 2) → Stesso su v2

Esercizio Cosa Stampa

```
template<class Functor>
vector<int> find_template(const vector<int>& v, Functor t) {
    vector<int> r;
    for(auto it = v.begin(); it != v.end(); ++it) if (t(*it)) r.push_back(*it);
    return r;
}

unsigned int find_1(const vector<int>& v,int k) {
  vector<int> w = find_template(v, [v,k] (int n) { return n>k; } );
  return w.size();
}

vector<int> find_2(const vector<int>& v) {
  return find_template(v, [v] (int n) { return n<v.size(); } );
}

vector<int> v1 = {3,6,4,6,2,5,-2,4,2}; vector<int> v2 = {-2,-6,4,4,2,5,0,4,2,3,2,0};
```

size() → Metodo che per definizione non può essere negativo (unsigned int)

Lambda → Catturo "v" e guardo che ogni "n" (funtore) sia minore di size() (purché sia positivo)

find_2(v1).size() → dim9 (8) → Tutti positivi tranne -2
find_2(v2).size() → dim12 (10) → Tutti positivi tranne -2 e -6

Esercizio Funzione Definire un template di funzione `template<class T> list<const iostream*> fun(vector<ostream*>&)` con il seguente comportamento: in ogni invocazione `fun(v)`, per ogni puntatore `p` elemento (di tipo `ostream*`) del vector `v`:

1. se `p` non è nullo e `*p` è un `fstream` che non è nello stato good (ovvero stato 0, con tutti i bit di errore spenti), allora `p` diventa nullo;
2. se `p` non è nullo e `*p` è uno `stringstream` nello stato good, allora `p` viene inserito nella lista che la funzione deve ritornare;
3. se la lista che la funzione deve ritornare è vuota allora la funzione solleva una eccezione di tipo `T`, altrimenti ritorna la lista.

```
template <class T> std::list<const iostream*> fun(std::vector <ostream*>& v){
        std::list<const iostream*> i;

        for(vector <ostream*>::iterator it = v.begin(); it ≠ v.end(); ++it){

                // p = ostream → fstream

                fstream *f = dynamic_cast<fstream*>(*it);

                // (a)
                if(f && f→good() == 0){
                        f = nullptr;
                }


                // se fosse stato const ostream?

                //(1a) cosa da persone sane 😀
                fstream *f = dynamic_cast<fstream*>(const_cast<ios*>(*it));

                //(2a) → se tipo costante = const_iterator 😳
                const fstream* f = dynamic_cast<const fstream*>(*it);

                (b)
                stringstream *s = dynamic_cast<stringstream*>(*it);

                if(s && s→good()){
                        i.push_back(s);
                }
        }

        if(i.isEmpty())
                throw T();
        else
                return i;
}
```

Si assuma che `Abs` sia una classe base astratta fissata. Definire un template di funzione `bool Fun(T1*, T2&)`, dove `T1` e `T2` sono parametri di tipo, con il seguente comportamento. Si consideri una istanziazione implicita `Fun(ptr,ref)` dove si suppone che i parametri di tipo `T1` e `T2` siano istanziati a tipi polimorfi. Allora `Fun(ptr,ref)` ritorna `true` se e soltanto se valgono le seguenti due condizioni:

1. I parametri di tipo `T1` e `T2` sono istanziati allo stesso tipo;
2. Siano `D1*` il tipo dinamico di `ptr` e `D2&` il tipo dinamico di `ref`; allora: (i) `D1` e `D2` sono lo stesso tipo e (ii) questo tipo è un sottotipo proprio della classe `Abs`.

Scrivere la risposta nel riquadro sotto.

```
#include <typeinfo> (typename → T (C++) In Java → generico)

template <class T1, class T2>
bool Fun(T1* ptr, T2& ref){
        if(typeid(T1) == typeid(T2) && typeid(*ptr) ==
        typeid(ref) && dynamic_cast<Abs*>(*ptr))
                return true;
        return false;
}
```

Esercizio Cosa Stampa

```
class A {
public:
  A() {cout<< " A() ";}
  ~A() {cout<< " ~A ";}
  A(const A& x) {cout<< " Ac ";}
  virtual const A* j() {cout<<" A::j "; return this;}
  virtual void k() {cout <<" A::k "; m();}
  void m() {cout <<" A::m "; j();}
};

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

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

class D: virtual public B {
public:
  D() {cout<< " D() ";}
  ~D() {cout<< " ~D ";}
  virtual void g() {cout <<" D::g ";}
  const B* j() {cout <<" D::j "; return this;}
  void k() const {cout <<" D::k "; k();}
  void m() {cout <<" D::m "; g(); j();}
};

class E: public C, public D  {
public:
  E() {cout<< " E() ";}
  ~E() {cout<< " ~E ";}
  E(const E& x) {cout<< " Ec ";}
  virtual void g() const {cout <<" E::g ";}
  const E* j() {cout <<" E::j "; return this;}
  void m() {cout <<" E::m "; g(); j();}
  D& n() final {cout <<" E::n "; return *this;}
};

A* p1 = new E(); B* p2 = new C(); A* p3 = new D(); B* p4 = new E();
const A* p5 = new D(); const B* p6 = new E(); const E* p7 = new E();
```

(p1->j()->k());

E::j C::k B::n

Cosa stampa ⟶ 2 soli casi d'errore

- O SI HA Errore Runtime (C::k() C::k() C::k() ... → stack overflow)

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

- OPPURE Undefined behavior → Non definito da standard

Tendenzialmente si ha con i cast nulli

Scrivere un programma consistente di esattamente tre classi `A`, `B` e `C` e della sola funzione `main()` che soddisfi le seguenti condizioni:

1. la classe `A` è definita come:

   ```
   class A { public: virtual ~A(){} };
   ```

2. le classi `B` e `C` devono essere definite per ereditarietà e non contengono alcun membro
3. la funzione `main()` definisce le tre variabili:

   ```
   A* pa = new A; B* pb = new B; C* pc = new C;
   ```

   e nessuna altra variabile (di alcun tipo)

4. la funzione `main()` può **utilizzare solamente** espressioni di tipo `A*`, `B*` e `C*`, **non può** sollevare eccezioni mediante una `throw` e **non può** invocare l'operatore `new`
5. il programma deve compilare correttamente
6. l'esecuzione di `main()` **deve provocare un errore run-time**.

ERRORE RUNTIME → CAST NULL...

```
class B: public A {};
class C: public A {};
int main() {/* ...*/ dynamic_cast<C&>(*pb);}
```

Soluzione

Dereferencing a NULL pointer is undefined behavior.

In fact the standard calls this exact situation out in a note (8.3.2/4 "References"):

> Note: in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a null pointer, which causes undefined behavior.

SE NON HO SBAGLIATO, QUESTO È UB => UNDEFINED BEHAVIOUR

Si considerino le seguenti definizioni.

```
class B {
private:
  vector<bool>* ptr;
  virtual void m() const =0;
};

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

class F: public D {
private:
  list<int*> l;
  int& ref;
  double* p;
public:
  void m() const {}
  // ridefinizione del costruttore di copia di F
};
```

Ridefinire il costruttore di copia della classe `F` in modo tale che il suo comportamento coincida con quello del costruttore di copia standard di `F`.

```
// copia
F(const F& f): D(f), l(f.l), ref(f.ref)
                p(f.p);

// assegnazione

F& operator=(const F& f){
        if(this != f){
                D::operator=(f);
                l = f.l;
                ref = f.ref;
                p = f.p;
        }
        return *this;
}
```

```
// clonazione

virtual F* clone() const{
        return new F(*this);
}

// distruzione

~F(){
        if (p) delete p;

        for(int i = 0; i < l.size(); i++){
                auto *v = l[i];
                delete v;
        }

}
```

```
class A {
  public:
    virtual ~A() {}
};

class B: public A {};      class C: public A {};      class D: public C {};

template<class T>
void Fun(A* pt){
  bool b=0;
  try{ throw T(*pt); }
  catch(B){cout << "B "; b=1;}
  catch(C){cout << "C "; b=1;}
  catch(D){cout << "D "; b=1;}
  catch(A){cout << "A "; b=1;}
  if(!b) cout << "NO ";
}

B b; C c; D d; A *pa1 = &b, *pa2 = &d;
B *pb1 = dynamic_cast<B*>(pa1);  B *pb2 = dynamic_cast<B*>(pa2);
```

```
01:  Fun(&c);
02:  Fun(&d);
03:  Fun(pa1);
04:  Fun(pa2);
05:  Fun(pb1);
06:  Fun(pb2);
07:  Fun<A>(pb1);
08:  Fun<A>(pa2);
09:  Fun<B>(pb1);
10:  Fun<C>(pa2);
11:  Fun<C>(&d);
12:  Fun<D>(pa2);
```

Le precedenti definizioni compilano senza provocare errori (con gli opportuni `#include` e `using`).
Per ognuna delle seguenti 12 istruzioni di invocazione della funzione `Fun` della tabella scrivere chiaramente nel foglio 12 righe con numerazione da 01 a 12 e per ciascuna riga:

- **NON COMPILA** se la compilazione dell'istruzione provoca un errore;
- **UNDEFINED BEHAVIOUR** se l'istruzione compila correttamente ma la sua esecuzione provoca un undefined behaviour o un errore a run-time;
- se l'istruzione compila correttamente e non provoca errori a run-time allora si scriva la stampa che l'esecuzione produce in output su `cout`; se non provoca alcuna stampa allora si scriva **NESSUNA STAMPA**.

```
Fun(Fun(pa2)); cout << endl; // "AA"
Fun(Fun(pb2)); cout << endl; // "BA"
Fun(Fun(pb1)); cout << endl; // "BA"
```

Esercizio Cosa Stampa

```
class B {
public:
  int x;
  B(int z=1): x(z) {}
  virtual void f() const {cout << x << " B::f() ";}
};

class C: virtual public B {
public:
  virtual void g() const {cout << "C::g() ";}
  virtual void h() const {cout << "C::h() ";}
};

class D: virtual public B {
public:
  virtual void f() const {cout << "D::f() ";}
};

class E: public C {
public:
  virtual void f() const {cout << "E::f() ";}
  virtual void h() const {cout << "E::h() ";}
};

class F: public E, public D {
public:
  F(): B(3) {}
  virtual void f() const {cout << x << " F::f() ";}
  virtual void g() const {cout << "F::g() ";}
};

void Fun(const vector<B*>& v) {
  auto it1 = v.begin();
  vector<B*>::const_iterator it2;
  C* q;
  for(int i=1 ; it1 != v.end(); ++it1, ++i) {
    std::cout << "#" << i << " ";
    (*it1)->f();
    it2 = it1 + 1;
    if(it2 != v.end() && typeid(**it1) == typeid(**it2)) (*it2)->f();
    q = dynamic_cast<C*>(*it1);
    if(q) {static_cast<C*>(q)->g(); q->h();}
    cout << endl;
  }
}

int main() {
  B b; C c; D d; E e; F f;
  vector<B*> v = {  &d, &d, &e, &e, &b, &b, &f, &f, &e, &f, &c, &c };
  Fun(v);
}
```

Le precedenti definizioni compilano correttamente ed il `main` esegue senza undefined behavior o errori run-time. Scrivere nell'apposito spazio relativo alla riga `#i` le stampe prodotte in output dall'iterazione `i`-esima del ciclo for della funzione `fun`, scrivendo **NESSUNA STAMPA** se in una iterazione non ci fossero stampe prodotte in output.

```
#1 .....
#2 .....
#3 .....
#4 .....
#5 .....
#6 .....
#7 .....
#8 .....
#9 .....
#10 .....
#11 .....
#12 .....
```