Laurea in Informatica – Programmazione ad Oggetti – Appello d'Esame 14/02/2023

Nome..... Cognome..... Matricola.....

Esercizio Cosa Stampa

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

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

class F: public E {
public:
  F() {cout<< " F() ";}
  ~F() {cout<< " ~F() ";}
  F(const F& x): B(x) {cout<< " Fc ";}
  void k() {cout <<" F::k "; g();}
  void m() {cout <<" F::m "; j();}
};

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

Queste definizioni compilano correttamente (con opportuni `#include` e `using`). Per ognuno dei seguenti statement scrivere nell'apposito spazio:

- **NON COMPILA** se la compilazione dello statement provoca un errore;
- **UNDEFINED** se lo statement compila correttamente ma la sua esecuzione provoca un undefined behaviour o un errore run-time;
- se lo statement compila ed esegue correttamente (senza undefined behaviour o errori 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**.

```
C* ptr = new F(f); ..........

p1->m(); ..........

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

(dynamic_cast<const F*>(p1->j()))->g(); ..........

p2->m(); ..........

(p2->j())->g(); ..........

p3->k(); ..........

(p3->n()).m(); ..........

(dynamic_cast<D&>(p3->n())).g(); ..........

p4->f(); ..........

p4->k(); ..........

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

(p5->n()).g(); ..........

(dynamic_cast<E*>(p6))->j(); ..........

(dynamic_cast<C*>(const_cast<B*>(p7)))->k(); ..........

delete p7; ..........
```

Esercizio Tipi

```
class A {
public:
  virtual ~A() = 0;
};
A::~A() = default;

class B: public A {
public:
  ~B() = default;
};

char F(const A& x, B* y) {
  B* p = const_cast<B*>(dynamic_cast<const B*> (&x));
  auto q = dynamic_cast<const C*> (&x);
  if(dynamic_cast<E*> (y)) {
    if(!p || q) return '1';
    else return '2';
  }
  if(dynamic_cast<C*> (y)) return '3';
  if(q) return '4';
  if(p && typeid(*p) != typeid(D)) return '5';
  return '6';
}
```

```
class C: virtual public B {};
class D: virtual public B {};
class E: public C, public D {};
```

```
int main() {
B b; C c; D d; E e;


cout << F(....,....) << F(....,....) << F(....,....) << F(....,....) << F(....,....)


     << F(....,....) << F(....,....) << F(....,....) << F(....,....) << F(....,....);
}
```

Si considerino le precedenti definizioni ed il `main()` incompleto. Definire opportunamente negli appositi spazi `....,....` le chiamate alla funzione `F` di questo `main()` usando gli oggetti locali `b`, `c`, `d`, `e`, `f` in modo tale che: (1) non vi siano errori in compilazione o a run-time; (2) le chiamate di `F` siano **tutte diverse** tra loro; (3) l'esecuzione produca in output **esattamente** la stampa **`6544233241`**.

Esercizio Costruttore

```
class A {
private:
  virtual void f() const =0;
  vector<int*>* ptr;
};
```

```
class D: virtual public A {
private:
  int z;
  double w;
};
```

```
class E: public D {
private:
  vector<double*> v;
  int* p;
  int& ref;
public:
   void f() const {}
   E(): p(new int(0)), ref(*p) {}
   // ridefinizione del costruttore di copia di E
};
```

Si considerino le precedenti definizioni. Ridefinire (senza usare la keyword `default`) nello spazio sottostante il costruttore di copia della classe `E` in modo tale che il suo comportamento coincida con quello del costruttore di copia standard di `E`.

```
ridefinizione del costruttore di copia di E
```