METODOLOGIE DI PROGRAMMAZIONE (Output)

Esame 01-02-2005

```
class Base {
public:
  Base() {
    std::cout << "Constructor Base::Base()" << std::endl;
  }
  virtual void f(int) {
    std::cout << "Base::f(int)" << std::endl;
  }
  virtual void f(double) {
    std::cout << "Base::f(double)" << std::endl;
  }
  virtual void g(int) {
    std::cout << "Base::g(int)" << std::endl;
  }
  virtual ~Base() {
    std::cout << "Destructor Base::~Base()" << std::endl;
  }
};

class Derived : public Base {
public:
  Derived() {
    std::cout << "Constructor Derived::Derived()" << std::endl;
  }
  void f(char c) {
    std::cout << "Derived::f(char)" << std::endl;
  }
  void g(int) {
    std::cout << "Derived::g(int)" << std::endl;
  }
  ~Derived() {
    std::cout << "Destructor Derived::~Derived()" << std::endl;
  }
};

int main() {
  Base b;
  Derived d;
  Base& rb = b;
  Base* pb = &d;
  std::cout << "=== 1 ===" << std::endl;
  b.f(1);
  rb.f('a');
  rb.g(1);
  std::cout << "=== 2 ===" << std::endl;
  d.f(1);
  d.f(1.0);
  d.g(3.3);
  std::cout << "=== 3 ===" << std::endl;
  pb->f(1.0);
  pb->f('a');
  pb->g(3.3);
  return 0;
}
```

ConstructorBase::Base()
Constructor Base::Base()
Constructor Derived::Derived()
ConstructorBase::Base() *//I puntatori e le reference non attivano il costruttore, nel caso dei puntatori solo nel caso del comando new, avviene la chiamata al costruttore.*
=== 1 ===
Base::f(int)
Base::f(int) *//conversione implicita, char int*
Base::g(int)
=== 2 ===
Derived::f(char)*//vado a vedere le funzioni di base solo se non è possibile utilizzare/convertire in derived* Derived::f(char)
Derived::g(int)
=== 3 ===
Base::f(double)
Derived::f(char)
Derived::g(int)
Destructor Derived::~Derived()
Destructor Base::~Base()
Destructor Base::~Base()

Esame 22-02-2005

```cpp
class ZooAnimal {
public:
    ZooAnimal() {
        std::cout << "Constructor ZooAnimal" << std::endl;
    }
    virtual void print() {
        std::cout << "ZooAnimal::print" << std::endl;
    }
    virtual ~ZooAnimal() {}
};
class Bear : virtual public ZooAnimal {
public:
    Bear() {
        std::cout << "Constructor Bear" << std::endl;
    }
    void print() {
        std::cout << "Bear::print" << std::endl;
    }
    virtual ~Bear() {}
};
class Raccoon : virtual public ZooAnimal {
    public:
    Raccoon() {
        std::cout << "Constructor Raccoon" << std::endl;
    }
    virtual ~Raccoon() {}
};
class Endangered {
public:
    Endangered() {
        std::cout << "Constructor Endangered" << std::endl
    }
    void print() {
        std::cout << "Endangered::print" << std::endl;
    }
    virtual ~Endangered() {}
};
class Panda : public Endangered, public Bear, public Racco
public:
    Panda() {
        std::cout << "Constructor Panda" << std::endl;
    }
    void print() {
        std::cout << "Panda::print" << std::endl;
    }
    virtual ~Panda() {}
};
int main() {
    Panda ying_yang;
    ying_yang.print();

    Bear b = ying_yang;
    b.print();

    ZooAnimal* pz = &ying_yang;
    pz->print();

    Endangered& re = ying_yang;
    re.print();
    return 0;
}
```

---Panda ying_yang---
Constructor ZooAnimal*//Prima le classi virtuali*
Constructor Edangered
Constructor Bear
Construction Raccon
Construction Panda
---ying_yang.print()---
Panda::print
---Bear b = ying_yang---
//non succede nulla perché non creo nulla ma semplicemente copio
---b.print()---
Bear::print
--- ZooAnimal* pz = &ying_yang---
//il puntatore non attiva i costruttori, solo con il comando new
---pz->print()---
Panda::print
---Endangered& re = ying_yang---
//la referenze non crea nulla
---re.print()---
Endangered::print*//referenziando un oggetto, cambio la classe mentre il puntatore non fa nulla*

Esame 20-09-2005

```
class Base {
public:
  Base() {
    std::cout << "Constructor Base::Base()" << std::endl;
  }
  Base(const Base&) {
    std::cout << "Constructor Base::Base(const Base&)" << std::endl;
  }
  virtual void f() {
    std::cout << "Base::f()" << std::endl;
  }
  virtual void g() {
    std::cout << "Base::g()" << std::endl;
  }
  void h() {
    std::cout << "Base::h()" << std::endl;
  }
  virtual ~Base() {
    std::cout << "Destructor Base::~Base()" << std::endl;
  }
};

class Derived : public Base {
public:
  Derived() {
    std::cout << "Constructor Derived::Derived()" << std::endl;
  }
  Derived(const Derived&)
    : Base() {
    std::cout << "Constructor Derived::Derived(const Derived&)" << std::endl;
  }
  void f() {
    std::cout << "Derived::f()" << std::endl;
  }
  void h() {
    std::cout << "Derived::h()" << std::endl;
  }
  ~Derived() {
    std::cout << "Destructor Derived::~Derived()" << std::endl;
  }
};

int main() {
  Base b;
  Derived d;
  std::cout << "=== 0 ===" << std::endl;
  Base& rb = b;
  Base* pb = &d;
  Base b2 = *pb;
  Base* pb2 = &b2;
  std::cout << "=== 1 ===" << std::endl;
  b.f();
  rb.f();
  rb.h();
  std::cout << "=== 2 ===" << std::endl;
  d.f();
  d.g();
  d.h();
  std::cout << "=== 3 ===" << std::endl;
  pb->f();
  pb2->f();
  pb->g();
  pb->h();
  pb2->h();
  std::cout << "=== 4 ===" << std::endl;
  return 0;
}
```

#Base b
Constructor Base::Base()
#Derived d
Constructor Base::Base()
Constructor Derived::Derived()
=== 0 ===
#Base b2 = *pb //è come se stessi usando una reference
Constructor Base::Base(const Base&)
=== 1 ===
b.f()
Base::f()
rb.f()
Base::f()
#rb.h()
Base::h()
=== 2 ===
#d.f()
Derived::f()
#d.g()
Base::g()
#d.h()
Derived::h()
=== 3 ===
#pb->f()
Derived::f()
#pb2->f()
Base::f() //puntatore a puntatore
#pb->g()
Base::g()
#pb->h()
Base::h()
#pb2->h()
Base::h()
=== 4 ===
Destructor Base::~Base()
Destructor Derived::~Derived()
Destructor Base::~Base()
Destructor Base::~Base()

Esame 06-02-2006

```
#include <iostream>
using namespace std;

class A {
public:
  A() { cout << "Constructor A::A()" << endl; }

  virtual void f(int) { cout << "A::f(int)" << endl; }
  void f(double) { cout << "A::f(double)" << endl; }

  void g(double) { cout << "A::g(double)" << endl; }

  virtual ~A() { cout << "Destructor A::~A()" << endl; }
};

class B {
public:
  B() { cout << "Constructor B::B()" << endl; }

  void f(int) { cout << "B::f(int)" << endl; }
  virtual void f(double) { cout << "B::f(double)" << endl; }

  virtual void g(int) { cout << "B::g(int)" << endl; }

  virtual ~B() { cout << "Destructor B::~B()" << endl; }
};

class D : public B, public A {
public:
  D() { cout << "Constructor D::D()" << endl; }

  void f(int) { cout << "D::f(int)" << endl; }

  using A::g;
  void g(int) { cout << "D::g(int)" << endl; }

  ~D() { cout << "Destructor D::~D()" << endl; }
};

void h(A a, B b, D& d) {
  a.g('a');
  B* pb = &b;
  pb->f(4);
  d.g(44);
}

int main() {
  D d;
  A& ra = d;
  B& rb = d;
  cout << "=== 1 ===" << endl;
  ra.f(1);
  ra.g(1);
  rb.f(1);
  rb.g(1);
  cout << "=== 2 ===" << endl;
  d.f(1.2);
  d.g(1);
  d.g(1.2);
  cout << "=== 3 ===" << endl;
  h(d, d, d);
  cout << "=== 4 ===" << endl;
  return 0;
}
```

Constructor B::B()
Constructor A::A()
Constructor D::D() //non viene creato più nulla perchè sono delle reference
=== 1 ===
D::f(int) //dovrei usare la classe A ma li c'è una virtual
A::g(double)
B::f(int)
D::g(double)
=== 2 ===
D::f(int)
D::g(int)
A::g(double)// con using vede entrambe le funzioni g e sceglie la migliore(conversione esatta)
=== 3 ===
// A e B sono creati ma come copia, manca il costruttore di copia nelle classi, per esempio *A(const A&){cout << "Constructor A::A(A)" << endl;}*
A::g(double)
B::f(int)
D::g(int)
Destructor A::~A()
Destructor B::~B() // A e B vengono eliminati perchè non sono stati passati per reference ma direttamente creati
=== 4 ===
Destructor D::~D()
Destructor A::~A()
Destructor B::~B()

Esame 27-02-2006

```
#include <iostream>
using namespace std;

class Base {
public:
  Base() { cout << "Costruttore Base" << endl; }
  virtual void foo(int) { cout << "Base::foo(int)" << endl; }
  virtual void bar(int) { cout << "Base::bar(int)" << endl; }
  virtual void bar(double) { cout << "Base::bar(double)" << endl; }
  virtual ~Base() { cout << "Distruttore Base" << endl; }
};

class Derived : public Base {
public:
  Derived() { cout << "Costruttore Derived" << endl; }
  void foo(int) { cout << "Derived::foo(int)" << endl; }
  void bar(int) const { cout << "Derived::bar(int)" << endl; }
  void bar(double) const { cout << "Derived::bar(double) const" << endl; }
  ~Derived() { cout << "Distruttore Derived" << endl; }
};

void g(Base b) {
  b.foo(5);
  b.bar(5.5);
}

int main() {
  Derived derived;
  Base base;
  Base& base_ref = base;
  Base* base_ptr = &derived;
  Derived* derived_ptr = &derived;
  cout << "=== 1 ===" << endl;
  base_ptr->foo(12.0);
  base_ref.foo(7);
  base_ptr->bar(1.0);
  derived_ptr->bar(1.0);
  derived.bar(2);
  cout << "=== 2 ===" << endl;
  base.bar(1);
  derived.bar(-1.0);
  derived.foo(0.3);
  base_ptr->bar('\n');
  cout << "=== 3 ===" << endl;
  g(*derived_ptr);
  return 0;
}
```

Costruttore Base
Costruttore Derived
Costruttore Base
=== 1 ===
Derived :: foo(int)
Base :: foo(int)
Base :: bar(double)
Derived :: bar(double) const
Derived :: bar(int)
=== 2 ===
Base :: bar(int)
Derived :: bar(double) const
Derived :: foo(int)
Base :: bar(int) //Essendo Virtual, può richiamare anche la funzione in Derived, ma la funzione in Base è la migliore
=== 3 ===
Base :: foo(int)
Base :: bar(double)
Distruttore Base
Distruttore Base
Distruttore Derived
Distruttore Base

Esame 16-06-2008

```
#include <iostream>

class Animale {
public:
  Animale() { std::cout << "Costruttore Animale" << std::endl; }
  Animale(const Animale&) { std::cout << "Copia Animale" << std::endl; }

  virtual Animale* clone() const {
    std::cout << "Clonazione non specificata" << std::endl;
    return new Animale(*this);
  }

  virtual void verso() const {
    std::cout << "Verso non specificato" << std::endl;
  }

  virtual ~Animale() { std::cout << "Distruttore Animale" << std::endl; }
};

class Cane : public Animale {
public:
  Cane() { std::cout << "Costruttore Cane" << std::endl; }

  void verso() { std::cout << "bau!" << std::endl; }

  ~Cane() { std::cout << "Distruttore Cane" << std::endl; }

  Cane* clone() const { return new Cane(*this); }
};

class Pesce : public Animale {
public:
  Pesce() { std::cout << "Costruttore Pesce" << std::endl; }

  void verso() const { std::cout << "(glu glu)" << std::endl; }

  ~Pesce() { std::cout << "Distruttore Pesce" << std::endl; }

  Pesce* clone() const { return new Pesce(*this); }
};

class Pescecane : public Pesce {
public:
  Pescecane() { std::cout << "Costruttore Pescecane" << std::endl; }
  void verso() const { std::cout << "(glubau!)" << std::endl; }
  ~Pescecane() { std::cout << "Distruttore Pescecane" << std::endl; }
};

int main() {
  Animale a;
  a.verso();
  Cane c;
  c.verso();
  std::cout << "=== 1 ===" << std::endl;

   Pescecane p;
   p.verso();
   std::cout << "=== 2 ===" << std::endl;
   Animale* pc = c.clone();
   Animale* pp = p.clone();
   std::cout << "=== 3 ===" << std::endl;
   pc->verso();
   pp->verso();
   std::cout << "=== 4 ===" << std::endl;
   delete pp;
   delete pc;
   std::cout << "=== 5 ===" << std::endl;
}
```

Costruttore Animale
Verso non specificato
Costruttore Animale
Costruttore Cane
Bau!
=== 1 ===
Costruttore Animale
Costruttore Pesce
Costruttore PesceCane
(glubau!)
=== 2 ===
Copia Animale
Copia Animale
=== 3 ===
Verso non specificato // stai attento al const(<3)
Glu glu
=== 4 ===
Distruttore Pesce
Distruttore Animale
Distruttore Cane
Distruttore Animale
=== 5 ===
Distruttore PesceCane
Distruttore Pesce
Distruttore Animale
Distruttore Cane
Distruttore Animale
Distruttore Animale

Esame 14-06-2010

```
#include <iostream>
using namespace std;

struct A {
  virtual void f(int)    { cout << "A::f(int)" << endl; }
  virtual void f(double) { cout << "A::f(double)" << endl; }

  virtual void g() { cout << "A::g(double)" << endl; }

  virtual ~A() { cout << "Destructor A::~A()" << endl; }
};

struct B : public A {
  void f(int) { cout << "B::f(int)" << endl; }
  virtual void f(double) const { cout << "B::f(double) const" << endl; }

  virtual void g(int) { cout << "B::g(int)" << endl; }

  ~B() { cout << "Destructor B::~B()" << endl; }
};

struct C : public B {
  void f(int) const { cout << "C::f(int) const" << endl; }

  void g(int) { cout << "C::g(int)" << endl; }

  ~C() { cout << "Destructor C::~C()" << endl; }
};

int main() {
  A* a = new A;
  B b;
  C c;
  A& ra_b = b;
  B& rb_b = b;
  A& ra_c = c;
  B& rb_c = c;
  cout << "=== 1 ===" << endl;
  ra_b.f(1);
  rb_b.g(1);
  ra_c.f(1);
  rb_c.g(1);
  cout << "=== 2 ===" << endl;
  static_cast<A*>(&b)->f(1.2);
  static_cast<A*>(&c)->f(1);
  static_cast<B*>(&c)->g(1.2);
  cout << "=== 3 ===" << endl;
  b.f(2);
  c.g(3);
  cout << "=== 4 ===" << endl;
}
```

=== 1 ===
B :: f(int)
B :: g(int)
B :: f(int)
C :: g(int)
=== 2 ===//controllo il const con il tipo migliore, partendo dal basso
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()

Esame 05-07-2022

```
#include <iostream>

struct A{
        A(){
                std::cout<<"Ctor A::A()"<<std::endl;
        }
        virtual void f(int){
                std::cout<<"A::f(int)"<<std::endl;
        }
        virtual void f(float){
                std::cout<<"A::f(float)"<<std::endl;
        }
        virtual void g(int){
                std::cout<<"A::g(int)"<<std::endl;
        }
        virtual ~A(){
                std::cout<<"Dtor A::~A()"<<std::endl;
        }
};

struct B : public A{
        B(){
                std::cout<<"Ctor B::B()"<<std::endl;
        }
        using A::f;
        virtual void f(char){
                std::cout<<"B::f(char)"<<std::endl;
        }
        virtual void g(int){
                std::cout<<"B::g(int)"<<std::endl;
        }
        ~B(){
                std::cout<<"Dtor B::~B()"<<std::endl;
        }
};
struct C : public B{
        virtual void f(char){
                std::cout<<"C::f(char)"<<std::endl;
        }
        ~C(){
                std::cout<<"Dtor C::~C()"<<std::endl;
        }
}
 int main(){
        A a;
        C c;
        A& ra = a;
        B* pb = &c;

        std::cout<<"=== 1 ==="<<std::endl;
        pb->f(1.0F); // aggiunta di F perché il compilatore segnala che la chiamata all'overload di f(double) è ambigua
        pb->f('a');
        pb->g(3.1415);
        std::cout<<"=== 2 ==="<<std::endl;
        c.f(1);
        c.f(1.0);
        c.g(3.1315)
        std::cout<<"=== 3 ==="<<std::endl;
        a.f(1);
        ra.f('a');
        ra.g(1);
        return 0;
 }
```

```
Ctor A::A()
Ctor A::A()
Ctor B::B()
=== 1 ===
A::f(float)
C::f(char)
B::g(int)
=== 2 ===
C::f(char)
C::f(char)
B::g(int)
=== 3 ===
A::f(int)
A::f(int)
A::g(int)
=== 4 ===
Dtor C::-C()
Dtor B::-B()
Dtor A::-A()
Dtor A::-A()
```