Ereditarietà:

La capacità di riusare codice di una classe (classe padre, superclasse) da parte di un'altra classe (classe figlia, sottoclasse, classe derivata).


```
int main() {
      enum Colore{rosso, arancio, giallo, verde, blu, viola};
      class Punto {       // classe interna!
          public:         int x; int y;
                          void muovi(int dx, int dy){x+=dx;y+=dy;}
      };
      class PuntoColorato %figlia di% Punto {
          public:         Colore color;
                          void setColor(Colore a){color=a;}
      };
      PuntoColorato k;
      k.x=3; k.y=7;       // variabili di istanza ereditate
      k.color=giallo;
      k.muovi(2,4);       // metodo ereditato
      k.setColor(blu);
      cout<<k.x<<" "<<k.y<<" "<<k.color<<endl;
}
```

Ereditarietà

```
int main() {
    enum Colore{rosso, arancio, giallo, verde, blu, viola};
    class Punto {
        public:     int x; int y;
                    void muovi(int dx, int dy){x+=dx;y+=dy;}
    };
    class PuntoColorato:Punto{
       public:      Colore color;
                    void setColor(Colore a){color=a;}
    };
    PuntoColorato k;
    k.x=3; k.y=7;
    k.color=giallo;
    k.muovi(2,4);
    k.setColor(blu);
    cout<<k.x<<" "<<k.y<<" "<<k.color<<endl;
}
```

OUTPUT:

```
[C++ Error] Project1.cpp(13): E2247
' Punto::x' is not accessible.
' Punto::muovi()' is not accessible.
```


Ereditarietà pubblica

```
int main() {
    enum Colore{rosso, arancio, giallo, verde, blu, viola};
    class Punto {
        public:     int x; int y;
                    void muovi(int dx, int dy){x+=dx;y+=dy;}
    };
    class PuntoColorato:public Punto{
       public:      Colore color;
                    void setColor(Colore a){color=a;}
    };
    PuntoColorato k;
    k.x=3; k.y=7;
    k.color=giallo;
    k.muovi(2,4);
    k.setColor(blu);
    cout<<k.x<<" "<<k.y<<" "<<k.color<<endl;
}
```

OUTPUT:

```
5 11 4
```

Method overriding

```
int main() {
        enum Colore{rosso, arancio, giallo, verde, blu, viola};
        class Punto {
            public:         int x; int y;
                            void muovi(int dx, int dy){x+=dx;y+=dy;}
        };
        class PuntoColorato:public Punto{
           public:          Colore color;
                            void setColor(Colore a){color=a;}
                            void muovi(int dx, int dy){
                                    x+=dx;y+=dy;color+=1;}
        };
        PuntoColorato z;
        z.muovi();
        z.Punto::muovi();
}
```


Costruttori e distruttori

```
int main() {
        enum Colore{rosso, arancio, giallo, verde, blu, viola};
        class Punto {
            public:         int x; int y;
                            Punto(){cout<<"creo Punto";}
                            ~Punto(){cout<<"distruggo Punto";}
        };
        class PuntoColorato:public Punto {
           public:  Colore color;
                    PuntoColorato(){cout<<"creo PuntoColorato ";}
                    ~ PuntoColorato(){cout<<"distruggo PuntoColorato";}
        };
        PuntoColorato k;
}
```

OUTPUT:
creo Punto
creo PuntoColorato
distruggo PuntoColorato
distruggo Punto

Costruttori e distruttori

```
int main() {
      enum Colore{rosso, arancio, giallo, verde, blu, viola};
      class Punto {
          public:        int x; int y;
                         Punto(){cout<<"creo Punto";}
                         ~Punto(){cout<<"distruggo Punto";}
      };
      class PuntoColorato:public Punto {
         public:  Colore color;
                  PuntoColorato(){cout<<"creo PuntoColorato ";}
                  ~ PuntoColorato(){cout<<"distruggo PuntoColorato";}
      };
      PuntoColorato * k=new PuntoColorato;
}
```

OUTPUT:
creo Punto
creo PuntoColorato
distruggo PuntoColorato
distruggo Punto

Costruttori

Ordine di chiamata:
1: costruttore della superclasse (senza parametri!)
2: costruttore della classe (con parametri eventuali)

Se la gerarchia è più lunga, si risale fino all'origine chiamando tutti i costruttori necessari!

Es. In B:public A C:public B la creazione di una istanza di C chiama (nell'ordine):
costruttore di A – costruttore di B – costruttore di C

Distruttori

Ordine di chiamata:
1: distruttore della classe
2: distruttore della superclasse

Se la gerarchia è più lunga, si risale fino all'origine chiamando tutti i distruttori necessari!

Es. In B:public A C:public B la distruzione di una istanza di C chiama (nell'ordine):
distruttore di C – distruttore di B – distruttore di A

MA: ATTENZIONE AGLI OGGETTI REFERENZIATI DA PUNTATORI!

Costruttori e distruttori

```
int main() {
      enum Colore{rosso, arancio, giallo, verde, blu, viola};
      class Punto {
          public:        int x; int y;
                         Punto(){cout<<"creo Punto";}
                         ~Punto(){cout<<"distruggo Punto";}
      };
      class PuntoColorato:public Punto {
         public:  Colore color;
                  PuntoColorato(){cout<<"creo PuntoColorato ";}
                  ~ PuntoColorato(){cout<<"distruggo PuntoColorato";}
      };
      Punto * k=new PuntoColorato;
      delete k;
}
```

OUTPUT:
creo Punto
creo PuntoColorato
distruggo Punto

STATIC BINDING

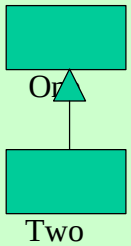
Distruttori virtuali

```
int main() {
    enum Colore{rosso, arancio, giallo, verde, blu, viola};
    class Punto {
        public:        int x; int y;
                       Punto(){cout<<"creo Punto";}
                       virtual ~Punto(){cout<<"distruggo Punto";}
    };
    class PuntoColorato:public Punto {
       public:  Colore color;
                PuntoColorato(){cout<<"creo PuntoColorato ";}
                ~ PuntoColorato(){cout<<"distruggo PuntoColorato";}
    };
    Punto * k=new PuntoColorato;
    delete k;
}
```

DYNAMIC BINDING

OUTPUT:
creo Punto
creo PuntoColorato
distruggo PuntoColorato
distruggo Punto


```
void directAssign(One x) {
  cout << "by assignment value is " << x.value() << endl;
}
void byReference(One & x) {
  cout << "by reference value is " << x.value() << endl;
}
void byPointer(One * x) {
  cout << "by pointer value is " << x->value() << endl;
}
int main()
{
  Two x;
  directAssign(x);
  byPointer(&x);
  byReference(x);
}
```



Virtual methods

```
#include <iostream.h>

class One {
  public:
    int value() {return 1;}
};

class Two:public One {
  public:
    int value() { return 2;}
};
```

```
by assignment value is 1
by pointer value is 1
by reference value is 1
```


Virtual methods

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

class One {
  public:
    virtual int value() {return 1;}
};

class Two:public One {
  public:
    int value() { return 2;}
};
```

```
by assignment value is 1
by pointer value is 2
by reference value is 2
```

Virtual methods

```
int main()
{
  Two x;
  One &a=x;
  One *b;
  One c;
  b=&x;
  c=x;
  cout << a.value() << b->value() << c.value() <<endl;
}
```

221

Ripasso:
Esempio con i fischietti

```
class Fischietto {
  public:
      void fischia();
      void fischia(int n);
      void cadi();
};
```

```
#include "Fischietto.h"
      void Fischietto::fischia() {
         cout<<"Fiii ";
         cout<<endl;
      }
      void Fischietto::fischia(int n){
        for (int k=0; k<n; k++)
          cout<<"Fiii ";
        cout<<endl;
      }
      void Fischietto::cadi() {
         cout<<"Toc ";
         cout<<endl;
      }
```



```
class FischiettoBitonale:public Fischietto{
  public:
     void fischia();
     void fischia(int n);
  };
```

Ereditarietà (Inheritance)

```
#include "FischiettoBitonale.h"
      void FischiettoBitonale::fischia() {
          cout<<"FuuuFiii ";
          cout<<endl;
      }
      void FischiettoBitonale::fischia(int n){
        for (int k=0; k<n; k++)
          cout<<"FuuuFiii ";
        cout<<endl;
      }
```

Method overriding

```
int main()
{
  Fischietto f;
  f.fischia();
  f.fischia(3);
  f.cadi();

  FischiettoBitonale fb;
  fb.fischia();
  fb.fischia(3);
  fb.cadi();
```

```
int tipo=0;
do {
  cout<<"Scegli [1:normale - 2:bitonale] :";
  cin>>tipo;
} while (tipo<1 || tipo>2);
Fischietto * x;
switch (tipo) {
  case 1: x=&f; break;
  case 2: x=&fb; break;
  default:
     cout<<"qui non deve mai passare!";
}
x->fischia();
x->fischia(4);
x->cadi();
return 0;
}
```



```
Fiii
Fiii Fiii Fiii
Toc

FuuuFiii
FuuuFiii FuuuFiii FuuuFiii
Toc

Scegli [1:normale - 2:bitonale] :2
Fiii
Fiii Fiii Fiii Fiii
Toc
```

STATIC BINDING

```
class Fischietto {
  public:
      virtual void fischia();
      void fischia(int n);
      void cadi();
};
```

```
#include "Fischietto.h"
      void Fischietto::fischia() {
         cout<<"Fiii ";
         cout<<endl;
      }
      void Fischietto::fischia(int n){
        for (int k=0; k<n; k++)
          cout<<"Fiii ";
        cout<<endl;
      }
      void Fischietto::cadi() {
         cout<<"Toc ";
         cout<<endl;
      }
```


Fiii
Fiii Fiii Fiii
Toc

FuuuFiii
FuuuFiii FuuuFiii FuuuFiii
Toc

Scegli [1:normale - 2:bitonale] :2
FuuuFiii ✍ DYNAMIC BINDING
Fiii Fiii Fiii Fiii ✍ STATIC BINDING
Toc

```
class Fischietto {
  public:
      virtual void fischia();
      virtual void fischia(int n);
      void cadi();
};
```

```
#include "Fischietto.h"
      void Fischietto::fischia() {
         cout<<"Fiii ";
         cout<<endl;
      }
      void Fischietto::fischia(int n){
        for (int k=0; k<n; k++)
          cout<<"Fiii ";
        cout<<endl;
      }
      void Fischietto::cadi() {
         cout<<"Toc ";
         cout<<endl;
      }
```



```
Fiii
Fiii Fiii Fiii
Toc

FuuuFiii
FuuuFiii FuuuFiii FuuuFiii
Toc

Scegli [1:normale - 2:bitonale] :2
FuuuFiii
FuuuFiii FuuuFiii FuuuFiii FuuuFiii
Toc
```

DYNAMIC BINDING

```
int main()
{
  Fischietto f;
  f.fischia();
  f.fischia(3);
  f.cadi();

  FischiettoBitonale fb;
  fb.fischia();
  fb.fischia(3);
  fb.cadi();
```

```
int tipo=0;
do {
   cout<<"Scegli [1:normale - 2:bitonale] :";
   cin>>tipo;
} while (tipo<1 || tipo>2);
Fischietto x;
switch (tipo) {
  case 1: x=f; break;
  case 2: x=fb; break;
  default:
     cout<<"qui non deve mai passare!";
}
x.fischia();
x.fischia(4);
x.cadi();
return 0;
}
```



```
Fiii
Fiii Fiii Fiii
Toc

FuuuFiii
FuuuFiii FuuuFiii FuuuFiii
Toc

Scegli [1:normale - 2:bitonale] :2
Fiii
Fiii Fiii Fiii Fiii
Toc
```

Il dynamic binding NON SI APPLICA alle variabili (solo a puntatori)

EREDITARIETA' 27

```
int main()
{
 Fischietto f;
 FischiettoBitonale f;
 int tipo=0;
  do {
     cout<<"Scegli [1:normale - 2:bitonale] :";
     cin>>tipo;
  } while (tipo<1 || tipo>2);
  switch (tipo) {
    case 1: esegui(f); break;
    case 2: esegui(fb); break;
    default: cout<<"qui non dev
  }
  int h;cin>>h;
  return 0;
}
```

```
void esegui(Fischietto x) {
  x.fischia();
  x.fischia(4);
  x.cadi();
}
```

```
Scegli [1:normale - 2:bitonale] :2
Fiii
Fiii Fiii Fiii Fiii
Toc
```

EREDITARIETA' 28

```
int main()
{
 Fischietto f;
 FischiettoBitonale f;
 int tipo=0;
  do {
     cout<<"Scegli [1:normale - 2:bitonale] :";
     cin>>tipo;
  } while (tipo<1 || tipo>2);
  switch (tipo) {
    case 1: esegui(f); break;
    case 2: esegui(fb); break;
    default: cout<<"qui non dev
  }
  int h;cin>>h;
  return 0;
}
```

```
void esegui(Fischietto & x) {
  x.fischia();
  x.fischia(4);
  x.cadi();
}
```

```
Scegli [1:normale - 2:bitonale] :2
FuuuFiii
FuuuFiii FuuuFiii FuuuFiii FuuuFiii
Toc
```

```
int main()
{
 Fischietto f;
 FischiettoBitonale f;
 int tipo=0;
  do {
     cout<<"Scegli [1:normale - 2:bitonale] :";
     cin>>tipo;
  } while (tipo<1 || tipo>2);
  switch (tipo) {
    case 1: esegui(&f); break;
    case 2: esegui(&fb); break;
    default: cout<<"qui non deve mai passare!";
  }
  int h;cin>>h;
  return 0;
}
```

```
void esegui(Fischietto * x) {
  x->fischia();
  x->fischia(4);
  x->cadi();
}
```

```
Scegli [1:normale - 2:bitonale] :2
FuuuFiii
FuuuFiii FuuuFiii FuuuFiii FuuuFiii
Toc
```


Costruttori e distruttori

```
#include <iostream.h>
using namespace std;
class Fischietto {
 public:
 Fischietto() { cout<<"creating Fischietto "<<endl; }
 ~ Fischietto () { cout<<"destroying Fischietto "<<endl; }
};

class FischiettoBitonale:public Fischietto {
 public:
 FischiettoBitonale() { cout<<"creating FischiettoBitonale "<<endl; }
  ~ FischiettoBitonale() { cout<<"destroying FischiettoBitonale "<<endl;}
};

int main()
{
       FischiettoBitonale p;
       return 0;
}
```

```
OUTPUT:
creating Fischietto
creating FischiettoBitonale
destroying FischiettoBitonale
destroying Fischietto
```

Costruttori e distruttori

```
#include <iostream.h>
using namespace std;
class Fischietto {
 public:
 Fischietto() { cout<<"creating Fischietto "<<endl; }
 ~ Fischietto () { cout<<"destroying Fischietto "<<endl; }
};

class FischiettoBitonale:public Fischietto {
 public:
 FischiettoBitonale() { cout<<"creating FischiettoBitonale "<<endl; }
  ~ FischiettoBitonale() { cout<<"destroying FischiettoBitonale "<<endl;}
};

int main()
{
  FischiettoBitonale *p = new FischiettoBitonale ;
 delete p;
  return 0;
}
```

OUTPUT:
creating Fischietto
creating FischiettoBitonale
destroying FischiettoBitonale
destroying Fischietto

Costruttori e distruttori

```
#include <iostream.h>
using namespace std;
class Fischietto {
 public:
 Fischietto() { cout<<"creating Fischietto "<<endl; }
 ~ Fischietto () { cout<<"destroying Fischietto "<<endl; }
};

class FischiettoBitonale:public Fischietto {
 public:
 FischiettoBitonale() { cout<<"creating FischiettoBitonale "<<endl; }
  ~ FischiettoBitonale() { cout<<"destroying FischiettoBitonale "<<endl;}
};

int main()
{
  Fischietto *p = new FischiettoBitonale ;
 delete p;
  return 0;
}
```

OUTPUT:
creating Fischietto
creating FischiettoBitonale
destroying Fischietto

Costruttori e distruttori

```
#include <iostream.h>
using namespace std;
class Fischietto {
 public:
 Fischietto() { cout<<"creating Fischietto "<<endl; }
 virtual ~ Fischietto () { cout<<"destroying Fischietto "<<endl; }
};

class FischiettoBitonale:public Fischietto {
 public:
 FischiettoBitonale() { cout<<"creating FischiettoBitonale "<<endl; }
  virtual ~ FischiettoBitonale()
                { cout<<"destroying FischiettoBitonale "<<endl;}
};

int main()
{
  Fischietto *p = new FischiettoBitonale ;
 delete p;
  return 0;
}
```

OUTPUT:
creating Fischietto
creating FischiettoBitonale
destroying FischiettoBitonale
destroying Fischietto