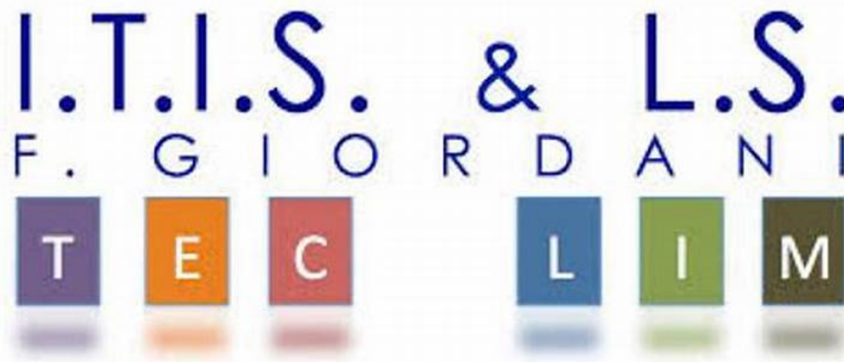


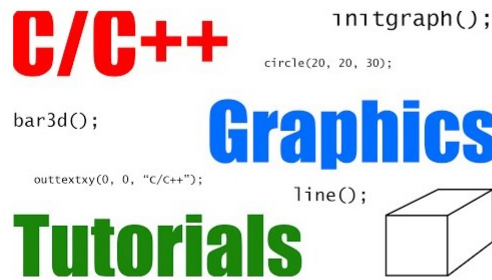
ITIS-LS “Francesco Giordani” Caserta

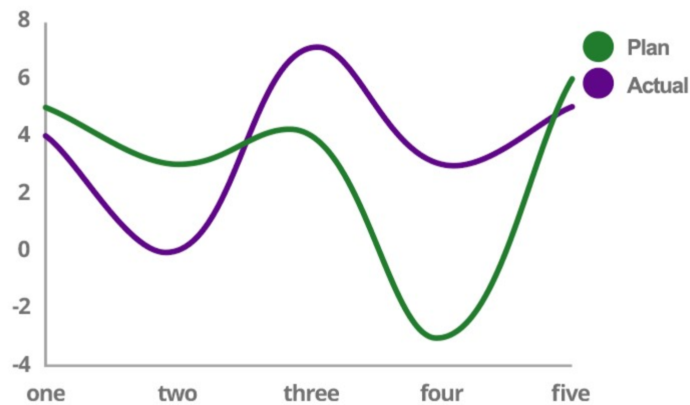
prof. Ennio Ranucci

a.s. 2019-2020

Semplici esercitazioni in modalità grafica C++

Animazioni e Classi derivate in C++





ITIS-LS “Francesco Giordani” Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell’esercizio: *es1*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica e animazione
Obiettivo del programma:
disegnare una linea e spostarla simulando un movimento verso il basso

```
#include<graphics.h>   // declarations for graphics library

#include<conio.h>      // for getch() function

#include<dos.h>

int main()

{

  // initialize the graphics system

  int pilotagrafico = DETECT, modografico;

  initgraph(&pilotagrafico, &modografico, "");

  int x1=100,x2=200,y1=100,y2=100;

  line (x1,y1,x2,y2);

  for(int i=1;i<150;i++)

  {

    setcolor(getbkcolor());

    line (x1,y1,x2,y2);

    y1=y1+1;

    y2=y2+1;

    setcolor(15);

    line (x1,y1,x2,y2);

    delay(100);

  }

  getch();              // Wait for keypress

  closegraph();          // Restore original screen

  return 0;

}
```

ITIS-LS “Francesco Giordani” Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell’esercizio: *es2*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica
Obiettivo del programma:
disegnare un segmento utilizzando la nuova funzione initwindow e putpixel

```
#include<graphics.h>


void segmento (int x1, int y1, int x2, int y2, int col);

int main( )

 {

   initwindow(500,500); //apre una finestra grafica 100x100

   segmento(100,100,200,100,12);

   getch();

   return 0;

}


void segmento (int x1, int y1, int x2, int y2, int col)

{

double x,y;

int i;

for (x=x1,y=y1,i=0;i<1000;i++)

   {

   putpixel((int) x, (int) y, col);

   x = x + (x2-x1)/1000.0;

   y = y + (y2-y1)/1000.0;

   }

}
```

ITIS-LS “Francesco Giordani” Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell’esercizio: *es3*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica
Obiettivo del programma:
disegnare un cerchio utilizzando la nuova funzione initwindow e circle

```
#include<graphics.h>


int main()

{

  initwindow(700,700,"MY First Program");

  circle(200,200,150);

  getch();

  return 0;

}
```

ITIS-LS “Francesco Giordani” Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell’esercizio: *es4*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica
Obiettivo del programma:
disegnare un rettangolo utilizzando la nuova funzione initwindow e putpixel

```
#include<graphics.h>


void segmento (int x1, int y1, int x2, int y2, int col);

void rettangolo(int xsup, int ysup, int xinf, int yinf, int col);

int main( )

 {

   initwindow(500,500); //apre una finestra grafica 100x100

   rettangolo(100,100,300,300,10);
```

```
   getch();
   return 0;
}


void segmento (int x1, int y1, int x2, int y2, int col)
{
double x,y;
int i;
for (x=x1,y=y1,i=0;i<1000;i++)
   {
   putpixel((int) x, (int) y, col);
   x = x + (x2-x1)/1000.0;
   y = y + (y2-y1)/1000.0;
   }
}


void rettangolo(int xsup, int ysup, int xinf, int yinf, int col)
{
segmento(xsup,ysup,xinf,ysup,col);
segmento(xsup,ysup,xsup,yinf,col);
segmento(xsup,yinf,xinf,yinf,col);
segmento(xinf,ysup,xinf,yinf,col);
}
```

ITIS-LS "Francesco Giordani" Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell'esercizio: *es5*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica
Obiettivo del programma:
disegnare un cerchio utilizzando la nuova funzione initwindow e putpixel

```
#include<graphics.h>

#include <math.h>

void cerchio(int x, int y, int r, int color);

int main( )

 {

   initwindow(500,500); //apre una finestra grafica 100x100

   cerchio(250,250,100,14);

   getch();

   return 0;

}


void cerchio(int x, int y, int r, int color)

{

static const double PI = 3.1415926535;

double i, angle, x1, y1;

for(i = 0; i < 360; i += 0.1)

   {

   angle = i;

   x1 = r * cos(angle * PI / 180);

   y1 = r * sin(angle * PI / 180);

   putpixel(x + x1, y + y1, color);

   }

}
```

ITIS-LS “Francesco Giordani” Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell’esercizio: *es6*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica
Obiettivo del programma:
disegnare cerchi concentrici

```
#include <graphics.h>


int main()

{

  int  schedagrafica= DETECT, risoluzione;

  int x = 320, y = 240, raggio;

  initgraph(&schedagrafica, &risoluzione, "");

  for ( raggio = 25; raggio <= 125 ; raggio = raggio + 40)  circle(x, y, raggio);

  getch();

  closegraph();

  return 0;

}
```

ITIS-LS "Francesco Giordani" Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell'esercizio: *es6*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica, Classi e classi derivate in c++
Obiettivo del programma:
Spostare un cerchio utilizzando la classe "locazione" e la sua derivata "punto"

```
#include <graphics.h>   // declarations for graphics library

#include <conio.h>      // for getch() function

class Locazione {

protected:        // allows derived class to access private data

  int X;

  int Y;


public:           // these functions can be accessed from outside

  Locazione(int InizX, int InizY);

  int daiX();

  int daiY();

};

class Punto : public Locazione {     // derived from class Location


  protected:

  bool Visibile;  // classes derived from Point will need access


public:

  Punto(int InizX, int InizY);     // constructor

  void Mostra();

  void Nascondi();

  bool Visibile_si();

  void Muovi_a(int NuovoX, int NuovoY);

};
```

```
// member functions for the Location class
Locazione::Locazione(int InizX, int InizY) {
  X = InizX;
  Y = InizY;
};

int Locazione::daiX(void) {
  return X;
};

int Locazione::daiY(void) {
  return Y;
};

// member functions for the Point class: These assume
// the main program has initialized the graphics system

Punto::Punto(int InizX, int InizY) : Locazione(InizX,InizY) {
  Visibile = false;             // make invisible by default
};

void Punto::Mostra(void) {
  Visibile = true;
  putpixel(X, Y, getcolor());     // uses default color
};

void Punto::Nascondi(void) {
  Visibile = false;
  putpixel(X, Y, getbkcolor()); // uses background color to erase
};

bool Punto::Visibile_si(void) {
```

```
  return Visibile;
};

void Punto::Muovi_a(int NuovoX, int NuovoY) {
  Nascondi();        // make current point invisible
  X = NuovoX;      // change X and Y coordinates to new location
  Y = NuovoY;
  Mostra();        // show point at new location
};

int main()
{
  // initialize the graphics system
  int pilotagrafico = DETECT, modografico;
  initgraph(&pilotagrafico, &modografico, "");

  // move a point across the screen
  Punto UnPunto(100, 50);   // Initial X, Y at 100, 50
  UnPunto.Mostra();          // Unpunto turns itself on
  getch();               // Wait for keypress
  UnPunto.Muovi_a(300, 150); // Unpunto moves to 300,150
  getch();               // Wait for keypress
  UnPunto.Nascondi();          // Unpunto turns itself off
  getch();               // Wait for keypress
  closegraph();          // Restore original screen
  return 0;
}
```

ITIS-LS "Francesco Giordani" Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell'esercizio: *es7*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica, Classi e classi derivate in c++
Obiettivo del programma:
spostare un pixel utilizzando la classe "locazione" e le sue derivate "punto" e "cerchio"

```
#include <graphics.h>   // declarations for graphics library

#include <conio.h>      // for getch() function

class Locazione {

protected:         // allows derived class to access private data

  int X;

  int Y;


public:           // these functions can be accessed from outside

  Locazione(int InizX, int InizY);

  int GetX();

  int GetY();

};

class Punto : public Locazione {      // derived from class Location


  protected:

  bool Visibile;  // classes derived from Point will need access


public:

  Punto(int InizX, int InizY);      // constructor

  void Mostra();

  void Nascondi();

  bool Visibile_si();

  void Muovi_a(int NuovoX, int NuovoY);

};
```

```
// member functions for the Location class
Locazione::Locazione(int InizX, int InizY) {
  X = InizX;
  Y = InizY;
};

int Locazione::GetX(void) {
  return X;
};

int Locazione::GetY(void) {
  return Y;
};

// member functions for the Point class: These assume
// the main program has initialized the graphics system

Punto::Punto(int InizX, int InizY) : Locazione(InizX,InizY) {
  Visibile = false;              // make invisible by default
};

void Punto::Mostra(void) {
  Visibile = true;
  putpixel(X, Y, getcolor());      // uses default color
};

void Punto::Nascondi(void) {
  Visibile = false;
  putpixel(X, Y, getbkcolor()); // uses background color to erase
};
```

```
bool Punto::Visibile_si(void) {
  return Visibile;
};

void Punto::Muovi_a(int NuovoX, int NuovoY) {
  Nascondi();        // make current point invisible
  X = NuovoX;      // change X and Y coordinates to new location
  Y = NuovoY;
  Mostra();        // show point at new location
};

// CIRCLE   A Circle class derived from Point


class Circonferenza : Punto {   // derived privately from class Point
                               // and ultimately from class Location
  int Raggio;          // private by default

public:
  Circonferenza(int InizX, int InizY, int InitRaggio);
  void Mostra(void);
  void Nascondi(void);
  void Espandi(int Espandi_di);
  void Muovi_a(int NuovoX, int NuovoY);
  void Contrai(int Contrai_di);
};

Circonferenza::Circonferenza(int InizX, int InizY, int InizRaggio)
                                                 : Punto(InizX,InizY)
{
  Raggio = InizRaggio;
};
```

```
void Circonferenza::Mostra(void)
{
  Visibile = true;
  circle(X, Y, Raggio);     // draw the circle
}

void Circonferenza::Nascondi(void)
{
  unsigned int TemporaneoColore;    // to save current color
  TemporaneoColore = getcolor();    // set to current color
  setcolor(getbkcolor());    // set drawing color to background
  Visibile = false;
  circle(X, Y, Raggio);     // draw in background color to erase
  setcolor(TemporaneoColore);      // set color back to current color
};

void Circonferenza::Espandi(int Espandi_di)
{
  Nascondi();                    // erase old circle
  Raggio += Espandi_di;          // expand radius
  if (Raggio < 0)               // avoid negative radius
    Raggio = 0;
  Mostra();                     // draw new circle
};

void Circonferenza::Contrai(int Contrai_di)
{
  Espandi(-Contrai_di);      // redraws with (Radius - Contrai_di)
};

void Circonferenza::Muovi_a(int NuovoX, int NuovoY)
```

```
{
  Nascondi();                  // erase old circle
  X = NuovoX;                  // set new location
  Y = NuovoY;
  Mostra();                    // draw in new location
};

main()                         // test the functions
{
  // initialize the graphics system
  int pilotagrafico = DETECT, modografico;
  initgraph(&pilotagrafico, &modografico, "");

  Circonferenza MiaCirconferenza(100, 200, 50);   // declare a circle object
  MiaCirconferenza.Mostra();                 // show it
  getch();                                   // wait for keypress
  MiaCirconferenza.Muovi_a(200, 250);       // move the circle (tests hide
                                            // and show also)
  getch();
  MiaCirconferenza.Espandi(50);             // make it bigger
  getch();
  MiaCirconferenza.Contrai(75);             // make it smaller
  getch();
  closegraph();
  return 0;
}
```

ITIS-LS "Francesco Giordani" Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell'esercizio: *es8*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica, Classi e classi derivate in c++
Obiettivo del programma:
spostare un pixel utilizzando la classe "locazione" e le sue derivate "punto" e "cerchio". Essendo uguali le funzioni "MoveTo" di Punto e di Cerchio, inseriamo in un comment la funzione MoveTo di Cerchio. Così la chiamata MyCircle.MoveTo(200, 250) risalirà la gerarchia delle classi ed utilizzera MoveTo dip unto generando lo spostamento del punto invece che del cerchio durante il run time;

```
#include <conio.h>      // for getch() function

#include <graphics.h>

class Location {

protected:         // allows derived class to access private data

  int X;

  int Y;


public:            // these functions can be accessed from outside

  Location(int InitX, int InitY);

  int GetX();

  int GetY();

};

class Point : public Location {     // derived from class Location


  protected:

  bool Visible;  // classes derived from Point will need access


public:

  Point(int InitX, int InitY);     // constructor

  void Show();

  void Hide();

  bool IsVisible();

  void MoveTo(int NewX, int NewY);
```

```
};


// member functions for the Location class
Location::Location(int InitX, int InitY) {
  X = InitX;
  Y = InitY;
};

int Location::GetX(void) {
  return X;
};

int Location::GetY(void) {
  return Y;
};

// member functions for the Point class: These assume
// the main program has initialized the graphics system

Point::Point(int InitX, int InitY) : Location(InitX,InitY) {
  Visible = false;            // make invisible by default
};

void Point::Show(void) {
  Visible = true;
  putpixel(X, Y, getcolor());     // uses default color
};

void Point::Hide(void) {
  Visible = false;
  putpixel(X, Y, getbkcolor()); // uses background color to erase
```

```
};

bool Point::IsVisible(void) {
  return Visible;
};

void Point::MoveTo(int NewX, int NewY) {
  Hide();       // make current point invisible
  X = NewX;     // change X and Y coordinates to new location
  Y = NewY;
  Show();       // show point at new location
};

// A Circle class derived from Point

class Circle : public Point { // derived from class Point
                              // and ultimately from class Location
  int Radius;         // private by default

public:
  Circle(int InitX, int InitY, int InitRadius);
  void Show(void);
  void Hide(void);
  void Expand(int ExpandBy);
  void Contract(int ContractBy);
  //void MoveTo(int NewX, int NewY);
};

// Circle constructor calls base Point constructor first
Circle::Circle(int InitX, int InitY, int InitRadius) : Point(InitX,InitY)
{
  Radius = InitRadius;
```

```
};

void Circle::Show()
{
  Visible = true;
  circle(X, Y, Radius);     // draw the circle using BGI function
}

void Circle::Hide()
{
  if (!Visible) return;     // no need to hide
  unsigned int TempColor;   // to save current color
  TempColor = getcolor();   // set to current color
  setcolor(getbkcolor());   // set drawing color to background
  Visible = false;
  circle(X, Y, Radius);     // draw in background color to erase
  setcolor(TempColor);      // set color back to current color
};

void Circle::Expand(int ExpandBy)
{
  bool vis = Visible;  // is current circle visible?
  if (vis) Hide();       // if so, hide it
  Radius += ExpandBy;    // expand radius
  if (Radius < 0)        // avoid negative radius
    Radius = 0;
  if (vis) Show();       // draw new circle if previously visible
};

inline void Circle::Contract(int ContractBy)
{
  Expand(-ContractBy);      // redraws with (Radius - ContractBy)
```

```
};
/*
void Circle::MoveTo(int NewX, int NewY) {
  Hide();        // make current point invisible
  X = NewX;      // change X and Y coordinates to new location
  Y = NewY;
  Show();        // show point at new location
};
 */
main()                    // test the functions
{
  // initialize the graphics system
  int graphdriver = DETECT, graphmode;
  initgraph(&graphdriver, &graphmode, "");

  Circle MyCircle(100, 200, 50);   // declare a circle object
  MyCircle.Show();               // show it
  getch();                       // wait for keypress
  MyCircle.MoveTo(200, 250);      // move the circle (tests hide
                                  // and show also)
  getch();
  MyCircle.Expand(50);           // make it bigger
  getch();
  MyCircle.Contract(75);         // make it smaller
  getch();
  closegraph();
  return 0;
}
```

OSSERVAZIONI

Cerchio.MoveTo è del tutto identico a Punto.MoveTo. Non è stato cambiato nulla, a parte copiare la routine con il qualificatore di Cerchio davanti all'identificatore MoveTo.

Questo esempio è la dimostrazione di un problema con oggetti e metodi impostati in questo modo. Fino ad ora, i metodi illustrati con riferimento ai tipi object Locazione, Punto e Cerchio sono di tipo statico (il termine statico è stato scelto per descrivere i metodi non virtuali. I metodi virtuali costituiscono in realtà una soluzione a questo problema, ma per capire la soluzione dovete prima comprendere il problema).

Il problema presenta le seguenti caratteristiche: a meno di non collocare una

copia del metodo MoveTo nel campo d'azione di Cerchio per escludere MoveTo di Punto, il metodo non funziona in modo corretto quando viene chiamato da un oggetto di tipo Cerchio. Infatti, se Cerchio invoca il metodo MoveTo di Punto, ciò che verrà spostato sullo schermo sarà un punto e non un cerchio. I cerchi potranno essere nascosti e tracciati dalle chiamate nidificate in Mostra e Nascondi, solo se Cerchio chiama una copia del metodo MoveTo definito nel suo campo d'azione.

La spiegazione si deve cercare nel modo in cui un compilatore risolve le chiamate dei metodi. Quando il compilatore compila i metodi di Punto, incontra prima Punto.Mostra e Punto.Nascondi e compila il codice per entrambi nel segmento codice. Un po' più in giù nel file troverà Punto.MoveTo che chiama sia Punto.Mosrra che Punto.Nascondi. Come avviene per qualsiasi chiamata di procedura, il compilatore sostituisce i riferimenti del codice sorgente a Punto.Mostra e Punto.Nascondi con gli indirizzi del loro codice generato nel segmento codice. Di conseguenza, quando viene chiamato il codice per Punto.Muovi_a, questi a sua volta chiama il codice per Punto.Mostra e Punto.Nascondi e tutto rimane in fase.

Il compilatore utilizza la seguente logica per risolvere le chiamate di metodo:

quando un metodo viene chiamato, il compilatore cerca come prima cosa un I

metodo con quel nome definito all'interno del tipo object. Il tipo Cerchio definisce i tipi di metodi chiamati Mostra, Nascondi, Espandi, Contrai e MoveTo. Qualora un metodo Cerchio dovesse chiamare uno di questi cinque metodi, il compilatore sostituirebbe la chiamata con l'indirizzo di uno dei metodi di Cerchio.

Se nessun metodo con quel nome è definito all'interno, di un tipo object, il compilatore risale fino al tipo antenato più vicino, cercando all'intemo di quel

tipo un metodo con il nome che è stato chiamato. Quando viene trovato un metodo con quel nome, l'indirizzo del metodo dell'antenato sostituisce il nome nel codice sorgente del metodo del discendente. Se non viene trovato nessun metodo con quel nome, il compilatore prosegue fino all'antenato successivo cercando il metodo con quel nome. Quando il compilatore raggiunge il primo tipo object in assoluto (principale), emette un messaggio d'errore per indicare che un tale metodo non è stato definito.

Tuttavia, dovete ricordare che quando viene trovato e utilizzato un metodo statico ereditato, il metodo che viene chiamato è esattamente quello che è stato definito e compilato per il tipo antenato. Quando il metodo dell'antenato chiama altri metodi, questi saranno i metodi dell'antenato, anche se il discendente possiede dei metodi che escludono quelli dell'antenato.

ITIS-LS “Francesco Giordani” Caserta
Anno scolastico: *2019/2020*
Classe 3^ sez.B spec. Informatica e telecomunicazioni
Data:
Numero progressivo dell’esercizio: *es9*
Versione: *1.0*
Programmatore/i:
Sistema Operativo: *Windows 10*
Compilatore/Interprete: *Logoit versione 6.4*
Obiettivo didattico:
Modalità grafica, Classi e classi derivate in c++ (ereditarietà), polimorfismo.
Obiettivo del programma:
spostare un pixel utilizzando la classe “locazione” e le sue derivate “punto” e “cerchio”. Essendo uguali le funzioni “MoveTo” di Punto e di Cerchio, inseriamo in un comment la funzione MoveTo di Cerchio ed utilizziamo la parola chiave “virtual” per realizzare il polimorfismo;

```
#include <conio.h>      // for getch() function

#include <graphics.h>

class Location {

protected:         // allows derived class to access private data

  int X;

  int Y;


public:           // these functions can be accessed from outside

  Location(int InitX, int InitY);

  int GetX();

  int GetY();

};

class Point : public Location {      // derived from class Location


  protected:

  bool Visible;  // classes derived from Point will need access


public:

  Point(int InitX, int InitY);      // constructor

  virtual void Show();

  virtual void Hide();

  bool IsVisible();

  void MoveTo(int NewX, int NewY);

};
```

```
// member functions for the Location class
Location::Location(int InitX, int InitY) {
  X = InitX;
  Y = InitY;
};

int Location::GetX(void) {
  return X;
};

int Location::GetY(void) {
  return Y;
};

// member functions for the Point class: These assume
// the main program has initialized the graphics system

Point::Point(int InitX, int InitY) : Location(InitX,InitY) {
  Visible = false;              // make invisible by default
};

void Point::Show(void) {
  Visible = true;
  putpixel(X, Y, getcolor());      // uses default color
};

void Point::Hide(void) {
  Visible = false;
  putpixel(X, Y, getbkcolor()); // uses background color to erase
};
```

```
bool Point::IsVisible(void) {
  return Visible;
};

void Point::MoveTo(int NewX, int NewY) {
  Hide();        // make current point invisible
  X = NewX;      // change X and Y coordinates to new location
  Y = NewY;
  Show();        // show point at new location
};

// A Circle class derived from Point

class Circle : public Point { // derived from class Point
                              // and ultimately from class Location
  int Radius;          // private by default

public:
  Circle(int InitX, int InitY, int InitRadius);
  void Show(void);
  void Hide(void);
  void Expand(int ExpandBy);
  void Contract(int ContractBy);
  //void MoveTo(int NewX, int NewY);
};

// Circle constructor calls base Point constructor first
Circle::Circle(int InitX, int InitY, int InitRadius) : Point(InitX,InitY)
{
  Radius = InitRadius;
};
```

```
void Circle::Show()
{
  Visible = true;
  circle(X, Y, Radius);     // draw the circle using BGI function
}

void Circle::Hide()
{
  if (!Visible) return;     // no need to hide
  unsigned int TempColor;   // to save current color
  TempColor = getcolor();   // set to current color
  setcolor(getbkcolor());   // set drawing color to background
  Visible = false;
  circle(X, Y, Radius);     // draw in background color to erase
  setcolor(TempColor);      // set color back to current color
};

void Circle::Expand(int ExpandBy)
{
  bool vis = Visible;  // is current circle visible?
  if (vis) Hide();       // if so, hide it
  Radius += ExpandBy;    // expand radius
  if (Radius < 0)        // avoid negative radius
    Radius = 0;
  if (vis) Show();       // draw new circle if previously visible
};

inline void Circle::Contract(int ContractBy)
{
  Expand(-ContractBy);      // redraws with (Radius - ContractBy)
};
```

```
/*
void Circle::MoveTo(int NewX, int NewY) {
  Hide();        // make current point invisible
  X = NewX;      // change X and Y coordinates to new location
  Y = NewY;
  Show();        // show point at new location
};
 */
main()                    // test the functions
{
  // initialize the graphics system
  int graphdriver = DETECT, graphmode;
  initgraph(&graphdriver, &graphmode, "");

  Circle MyCircle(100, 200, 50);   // declare a circle object
  MyCircle.Show();               // show it
  getch();                       // wait for keypress
  MyCircle.MoveTo(200, 250);      // move the circle (tests hide
                                  // and show also)
  getch();
  MyCircle.Expand(50);           // make it bigger
  getch();
  MyCircle.Contract(75);         // make it smaller
  getch();
  closegraph();
  return 0;
}
```

Fino ad ora gli unici metodi analizzati sono quelli statici. Essi sono tali per lo stesso motivo per cui le variabili statiche sono statiche: il compilatore le alloca e risolve tutto ciò che ad esse si riferisce durante la fase di compilazione(compile time).

In alcune circostanze possono non costituire il modo migliore per gestire i metodi.

*Problemi simili a quello descritto in **Pix3.cpp** sono dovuti alla soluzione nella fase di compilazione dei riferimenti dei metodi. Un modo per evitarli è quello di essere dinamici, risolvendo questi riferimento nella fase di esecuzione, perchè questo avvenga occorrono alcuni speciali meccanismi, e C++ li offre nel suo supporto dei metodi virtuali.*

- *IMPORTANTE*

*I metodi virtuali implemenrano uno strumento molto potente per la generalizzazione chiamato polimorfismo. Polimorfismo deriva dal greco e significa "con più forme"; e si tratta proprio di questo: **un modo per dare un nome ad un'azione che è condivisa lungo tutta una gerarchia di oggetti**, in cui ogni oggetto nella gerarchia implementa l'azione nel modo ad esso appropriato.*

La semplice gerarchia di figure grafiche come quelle che sono state descritte, rappresenta un ottimo esempio del polimorfismo, implementato per mezzo dei metodi virtuali.

Ogni tipo object nella nostra gerarchia rappresenta sullo schermo un tipo di figura diversa: un punto o un cerchio. E' certamente logico affermare che potete mostrare un punto o un cerchio sullo schermo. In seguito, se decidete di definire degli oggetti che definiscono altre figure, quali linee, quadrati, archi e così via, potete scrivere un metodo per ciascuna per visualizzare quell'oggetto sullo schermo. Secondo il nuovo modo di pensare orientato agli oggetti, potremmo affermare che tutti questi tipi di figure grafiche sono in grado di apparire sullo schermo. E questa caratteristica è condivisa da tutti.

Quello che invece è diverso per ogni tipo object, è il modo in cui deve apparire allo schermo. Un punto viene tracciato con una routine di plotting per punti che non richiede altro che una locazione X,Y e forse un valore di colore. La visualizzazione di un cerchio richiede una routine grafica completamente separata che non si limiti a considerare X, Y, ma anche un raggio.

E' possibile mostrare qualsiasi figura grafica, ma il meccanismo per mostrare ciascuna di esse dovrà essere specifico per ogni figura. Un'unica parola, "mostrare", viene utiiizzata per mostrare (letteralmente) più forme.

Quest'ultima frase è un ottimo esempio per spiegare cos'è il polimorfismo. E il C++ lo esegue con i metodi virtuali.

COLLEGAMENTO ANTICIPATO (EARLY BINDING)

E COLLEGAMELLTO RITARDATO (LATE BINDING)

La differenza che esiste tra una chiamata di metodo statico e una di metodo virtuale, può essere paragonata alla differenza che esiste tra una decisione presa subito e una rimandata. Quando codificate una chiamata di un metodo statico, è come dire al compilatore: "Sai cosa voglio. Chiamalo." Quando effettuate una chiamata di un metodo virtuale, è come dire al compilatore:

"Non sai quello che voglio -- per ora. Quando verrà il momento chiedi l'istanza" .

*Considerate questa metafora nei termini del problema MoveTo . Una chiamata a **Cerchio.MoveTo** cerca l'implementazione più vicina di MoveTo lungo la gerarchia dell'oggetto. In questo caso, Cerchio.MoveTo continuerebbe a chiamare la definizione di Punto di MoveTo, poichè Punto è il più vicino a Cerchio nella gerarchia. Supponendo che nessun tipo discendente abbia definito il proprio MoveTo perchè escludesse MoveTo di Punto, qualsiasi tipo discendente di Punto continuerà a chiamare la stessa implementazione di MoveTo. La decisione viene presa durante la fase di compilazione e non occorre fare altro.*

La situazione, tuttavia, cambia quando MoveTo chiama Mostra. Poichè ogni tipo figura possiede una propria implementazione di Mostra, quella chiamata da MoveTo dipenderà esclusivamente dalla istanza object che ha chiamato MoveTo in origine. Questo spiega il motivo per cui la chiamata al metodo Mostra all'interno dell'implementazione di MoveTo rappresenta una decisione rimandata: quando viene compilato il codice per MoveTo

non è possibile decidere quale Mostra chiamare, in quanto questa informazione non è disponibile nella fase di compilazione. Di conseguenza, la decisione dovrà essere rimandata alla fase di esecuzione, quando sarà possibile interrogare l'istanza object che chiama MoveTo.

Il processo mediante il quale le chiamate ai metodi statici vengono risolte in maniera non ambigua in un metodo singolo dal compilatore durante il tempo di compilazione viene chiamato collegamento anticipato (early binding).

Durante questo processo il chiamante e il chiamato vengono collegati alla prima opportunità, cioè nella fase di compilazione. Nel caso del collegamento ritardato (late binding), non è possibile collegare il chiamante e il chiamato durante la fase di compilazione, per cui viene attuato un particolare meccanismo per collegarli in seguito, quando viene realmente effettuata la chiamata.

Quando il compilatore incontra un metodo dichiarato come virtuale non può risolvere immediatamente tutti i riferimenti ad esso relativi e ha quindi bisogno di costruirsi una sorta di indice dei metodi virtuali presenti, per risolvere i riferimenti durante l'esecuzione del programma.

Questo indice è costituito dal campo VMT (Virtual Method Table, Tabella dei Metodi Virtuali), un campo dell'oggetto che il programmatore non vede e a cui non accede mai in modo diretto, ma che il compilatore aggiunge automaticamente alla definizione dell'oggetto quando questo contiene dei metodi virtuali.

ll campo VMT, la cui lunghezza è sempre di 16 bit, viene inserito subito

dopo i campi definiti esplicitamente per l'oggetto; per l'oggetto InsettoGenerico, definito come:

```
class insettoGenerico
   char[25]  Nome;
   int colore;
   int PosX;
   int PosY;
   insettogenerico();
   Posiziona(int X,Y);
   virtual Disegna (int ColoreDisegno);
   virtual Disegna2 (int ColoreDisegno);
```

la rappresentazione interna all'elaboratore prevederà il seguente ordine di

occupazione della memoria:

Nome

Colore

PosX

PosY

VMT

Supponendo che un oggetto discendente denominato InsettoNuovo

abbia la seguente struttura:

```
class  InsettoNuovo: public InsettoGenerico
   char[121] NomeScientifico
   Disegna (int ColoreDisegno):
   Disegna2 (int ColoreDisegno);
```

Si avrà il seguente ordine di occupazione della memoria:

Nome

Colore

PosX

PosY

VMT

NomeScientifico

Si noti che l'oggetto figlio estende il progenitore definendo un nuovo

campo NomeScientifico; questo campo viene posto in memoria in una locazione successiva a quella di VMT.

ll campo VMT del record, come tutte le variabili e campi sia di oggetti che di record, non viene inizializzato dal compilatore. Perchè esso contenga un valore significativo, deve essere inizializzato esplicitamente attraverso un metodo costruttore.

Lo scopo del campo VMT è semplice: esso contiene l'indirizzo di memoria dal quale inizia la tabella dei metodi virtuali (VMT) dell'oggetto. Nella tabella, che è memorizzata nel segmento dati riservato ai programmi, è contenuto il codice per i diversi metodi virtuali che appartengono al tipo di oggetto del campo VMT.

Diverse istanze di uno stesso tipo di oggetto condividono la stessa tabella

dei metodi virtuali, mentre esiste una sola tabella per ogni tipo di oggetto. per contro due oggetti dichiarati di tipo diverso non possono avere la stessa tabella anche se sono identici come struttura.

ll compilatore usa quindi l'informazione contenuta nel campo VMT per

andare a ripescare, nell'area dei programmi, i diversi metodi virtuali associati all'oggetto e risolvere i riferimenti tra metodi diversi solo al momento della compilazione. Si tratta di un metodo efficiente e agile per mantenere traccia dei rferimenti e collegarli solo al momento del bisogno in modo condizionale, a seconda dell'oggetto chiamante.

Questa tecnica comporta evidentemente degli inconvenienti: l'aggiunta di

un campo nascosto allunga un oggetto che faccia uso di metodi virtuali. Si

tratta di un appesantimento molto modesto in realtà, pari in ogni caso a due soli byte (16 bit).

Inoltre il compilatore deve consultare il campo VMT di un oggetto e poi

scorrere la tabella dei metodi virtuali alla ricerca del metodo necessario momento dell'esecuzione: questo processo rallenta, sia pur di poco, il programma in confronto all'utilizzo di metodi statici.

Si tratta del prezzo che si deve pagare (e non è molto, in verita !) per poter

usufruire de11a notevole flessibilità di definizione e di impiego degli oggetti messa a disposizione dall'uso dei metodi virtuali.

OOP

La programmazione strutturata ha rappresentato il primo tentativo di adottare tecniche di programmazione sovrapposte a un linguaggio di tipo tradizionale che potessero massimizzare l'efficienza nella stesura del software. Alla base della programmazione strutturata vi è il concetto di spezzare un programma complesso in unita più semplici e autoconsistenti.

Questo approccio, introdotto da Niklaus Wirth, offre senz'altro dei vantaggi, specie se associato con un linguaggio come il Pascal (anch'esso ideato da Wirth) che consente di applicare al meglio i dettami della programmazione strutturata.

La programmazione a oggetti (o anche programmazione orientata agli oggetti, traduzione del termine inglese Object Oriented Programming, abbreviato in 00P) rappresenta un modo di programmare radicalmente nuovo, che pone l'accento sul rapporto tra il programmatore e l'utente invece che tra programmatore e codice generato.

In sostanza, nella programmazione a oggetti invece di sbucciare il problema reale per poterlo rappresentare attraverso una serie di istruzioni elementari, si cerca di rappresentarlo, con tutte le sue caratteristiche e i suoi comportamenti specifici, all'interno del programma per farlo recitare, come se fosse un attore sul palcoscenico.

Per offrire una immagine intuitiva della differenza che esiste tra la pro- grammazione tradizionale e quella orientata agli oggetti, si può confrontare la rappresentazione che un'arancia, poniamo, potrebbe avere nell'uno e nell'altro modo.

Per un programma tradizionale, un'arancia è un insieme di spicchi, avvolti

da una sottile pellicola, ricoperti da una buccia; per la programmazione ad

oggetti è un'arancia, cioè un'entità elementare caratterizzata da alcune pro-

prietà (succosità, dolcezza, dimensioni, peso) che possono essere diverse da un esemplare all'altro, ma che tutte le arance possiedono. L'arancia ha anche un suo comportamento caratteristico: se viene spremuta secerne un succo acido e dolce (cosa che per esempio una mela non fa).

ll concetto generale di arancia (la categoria concettuale delle arance, se

si vuole) è l'oggetto che possiede degli attributi; all'interno del programma la rappresentazione di una certa particolare arancia (che pesa 125 grammi, ha un contenuto in succo del 15% e così via) costituisce un'istanza dell'oggetto arancia. Inoltre, per la programmazione ad oggetti I'arancia appartiene in modo naturale a una categoria più ampia, quella dei frutti, di cui condivide alcune proprietà (per esempio il peso e il contenuto di zucchero), pur avendone altre che la caratterizzano (per esempio l'acidità).

E' possibile quindi definire una categoria di oggetti più generale, quella dei frutti caratterizzata da proprietà comuni (peso, colore, contenuto zuccherino, stagione di raccolta, durata in magazzino) e da operazioni che su di essi possono compiere (spremere, far maturare artificialmente, tenere in magazzino in condizioni idonee).

L'oggetto arancia diviene a questo punto uno dei possibili figli dell'oggetto

genitore frutto, di cui eredita tutte le proprietà, senza che sia necessario ridefinirle esplicitamente ogni volta. E' evidente il risparmio di tempo e la riduzione degli errori possibile nel momento in cui, da una stessa categoria frutti si fanno derivare tutte le varietà di frutta esistenti: basta definire e verificare una volta sola campi e metodi dell'oggetto frutto per averli disponibili per decine di tipi di oggetti diversi, come mele, arance, pesche, banane e così via. Inoltre gli oggetti possono essere modificati rispetto al genitore da cui derivano, per cui al frutto arancia sarà possibile associare

il nuovo campo "acidità" che non ha senso per le banane. Questa prerogativa permette di adattare l'oggetto figlio alle specifiche esigenze che lo riguardano direttamente, pur conservando tutte le proprietà che ha ereditato dal genitore.

Inoltre, lo stesso tipo di azione (spremere, per esempio) potrà avere modi

di attuazione completamente diversi per ciascun frutto: l'arancia verrà spre

muta con lo spremiagrumi, mentre la mela dovrà essere centrifugata e la

banana frullata. Questa varietà di modi di realizzare una stessa operazione,

concettualmente identica, prende il nome nella programmazione ad oggetti di polimorfismo ed è una delle caratteristiche che conferiscono una flessibilità straordinaria a questo nuovo approccio alla programmazione

La programmazione a oggetti e i linguaggi object-oriented come C++ rendono disponibili al programmatore gli strumenti per poter realizzare questa astrazione, e consentono di realizzare software più modulare e

quindi più facile da riutilizzare.

Nella programmazione a oggetti è possibile costruire programmi com-

plessi assemblando moduli elementari compatti e chiusi, gli oggetti, dei quali è necessario conoscere solo come si comportano e quali messaggi devono scambiare con il mondo esterno per poter funzionare correttamente.

Notate l'analogia con i circuiti integrati modulari che hanno consentito lo

sviluppo dell'elettronica digitale: forse anche lo sviluppo del software si sta

muovendo lungo la strada giusta.

LESSICO

Le caratteristiche che identificano un ambiente di programmazione orien-

tato agli oggetti sono:

*-**L'incapsulamento**: l'oggetto contiene al suo interno sia dei campi che lo*

caratterizzano (nel caso dell'arancia i campi sono il colore, la succosità, il

peso e così via) sia i metodi, cioè le procedure che ne descrivono il com-

portamento (per esempio la capacità di secernere un succo se spremuta)

e che consentono di agire sui suoi campi:

***-L'eredità**: gli oggetti sono organizzati in categorie con una dipendenza*

gerarchica e oggetti di livello gerarchico inferiore possiedono le proprietà

che ereditano da oggetti ad essi superiori. L'arancia appartiene alla categoria gerarchicamente superiore dei frutti, e come tale ha un certo contenuto zuccherino, a differenza per esempio dei membri della categoria

delle verdure;

***-Il polimorfismo**: consente di definire una sola volta una certa azione (per*

esempio lo spremere) per tutti gli oggetti appartenenti a una certa gerarchia, ma consente di definire per ogni categoria di oggetti una variante del metodo, in base alle caratteristiche proprie di ciascun oggetto. Nella nostra analogia ortofrutticola, il metodo "estrazione del succo" diviene "spremere" per le arance, i limoni ed i pompelmi, mentre può essere "centrifugare" per frutti meno succosi, come le pere e le mele.

INCAPSULAMENTO

Con il termine incapsulamento si indica la compresenza, all'interno dell'og-

getto, sia dei dati (i campi) sia delle procedure che operano su questi dati (i metodi). L'incapsulamento porta con sé dei benefici (il principale dei quali è di tenere sempre allineati dati e procedure che operano su di essi) ma anche alcuni vincoli.

Una delle regole della buona programmazione orientata agli oggetti

impone di rendere totale l'incapsulamento. Questo significa che il programmatore deve rendere inutile l'accesso diretto ai campi degli oggetti, predisponendo dei metodi che accedano ad essi, e prevedendo dei metodi che eseguano tutte le elaborazioni che si ritiene opportuno compiere su questi campi.

Solo in questo modo è infatti possibile sfruttare appieno le caratteristiche

dell'incapsulamento e in particolare essere certi che le procedure che ope-

rano sui campi siano perfettamente adatte ad essi. Questa garanzia viene

meno se si lascia libero il programmatore di intervenire direttamente sui campi in punti che non siano interni ai metodi dell'oggetto. Inoltre l'accesso ai campi attraverso i metodi nasconde all'utente dell'oggetto sia il funzionamento dei metodi sia la struttura interna dei campi, a tutto vantaggio della chiarezza di impiego e della riusabilità degli oggetti.

EREDITARIETA'

La definizione di una gerarchia di oggetti costituisce una parte

importante nella progettazione di un sistema realizzato con un linguaggio

orientato agli oggetti.

Per maggiore chiarezza, si chiamano discendenti diretti quelli che deri-

vano dal progenitore, mentre tutti gli altri oggetti collegati da rapporti di parentela più lontani sono definiti genericamente progenitori o discendenti a seconda del loro ruolo. Si noti che mentre un progenitore può avere uno o più discendenti diretti, un dato discendente può avere uno ed un solo progenitore diretto.

L'ereditarietà è una delle caratteristiche degli oggetti che contribuiscono in

misura maggiore a cambiare il modo di programmare. Una situazione molto comune è quella in cui si deve realizzare una applicazione che differisce di poco da un'altra applicazione realizzata in precedenza. In questo caso utilizzando strumenti tradizionali di programmazione si deve esaminare tutto il codice, apportare le piccole modifiche richieste e poi verificare di nuovo tutto il software. Ciò che accade comunemente è che la modifica apportata per esempio alla

struttura dei dati (l'equivalente dell'aggiunta di un campo) richiede di modificare una procedura già scritta per includere il nuovo campo.

Da una semplice operazione come questa possono derivare sciagure impensabili: si vanno a toccare variabili a cui non si era pensato, si introducono errori di varia natura anche su parti del codice già ampiamente verificate e così via.

Utilizzando gli oggetti e la proprietà di ereditarietà questi problemi vengono facilmente aggirati: si definisce un nuovo oggetto per la nuova applicazione che si vuole realizzare come "figlio" di un oggetto dell'applicazione che si vuole assumere come modello e modificare. A questo punto l'oggetto figlio possiede tutti i campi e i metodi già definiti e collaudati per l'oggetto genitore;

si aggiungono eventuali campi e tutti i metodi necessari (sia metodi richiesti per gestire i nuovi campi, sia metodi per elaborare in modo nuovo i campi esistenti). Il risultato è un oggetto che risponde ai requisiti della nuova applicazione, per il quale si devono verificare esaustivamente solo i metodi aggiunti o modificati e che può essere immediatamente utilizzato nella nuova applicazione.

Ci si può domandare quale sia l'efficienza di un metodo come quello

descritto: in fondo, il nuovo oggetto si porta dietro dei metodi ereditati dall'oggetto genitore che però potrebbe non utilizzare mai. Questi metodi "inutili" non corrono il rischio di appesantire il programma senza portare nessun beneficio?

Su questo argomento interviene l'efficienza e la buona ingegneria del software : il linker di C++ (il programma che combina il modulo compilato con tutti i moduli di libreria necessari per ottenere un programma eseguibile) elimina dal programma eseguibile tutti i metodi che non vengono chiamati almeno una volta all'interno del programma. In questo modo i metodi ereditati e non utilizzati non vengono nemmeno inseriti nel programma finale, che viene distribuito ai clienti: non vi è nè appesantimento delle dimensioni del programma, nè rallentamento dovuto alla presenza di procedure non necessarie.

Nella programmazione orientata agli oggetti ci si deve porre il problema di

quale oggetto deve essere definito come progenitore e quali devono essere i figli. Una regola generale è la seguente:

l'oggetto progenitore deve essere quello con il minor contenuto di informazione

In questo modo sarà facile ottenere degli oggetti figli a cui aggiungere

campi e metodi, conservando la base comune ma differenziandoli per addizioni progressive.

Questo modo di procedere non necessariamente riflette le categorizza-

zioni imposte in diversi campi della scienza; in genere riflette piuttosto bene tutti gli aspetti evolutivi dei viventi ma può non essere altrettanto efficace nel rappresentare situazioni più artificiali.

Nel campo della geometria piana, per esempio, è possibile definire diversi tipi di relazione tra le figure geometriche. Si può partire da un generico poligono che viene via via specializzato per ottenere i diversi casi di figure regolari.

In questo modo, dal poligono generico si ricaveranno i poligoni regolari

imponendo che gli angoli siano tutti uguali; un esagono sarà poi quel poligono i cui angoli valgono 120', mentre un quadrato avrà angoli di 90' e un triangolo equilatero avrà angoli di 60'. In questo modo però si perdono informazioni:

mentre per disegnare un poligono irregolare generico è necessario fornire dati come la posizione dei vertici (o in alternativa la lunghezza di ciascun lato e l'angolo che esso forma con il lato adiacente), per definire il triangolo equilatero basta solo indicare la lunghezza di un lato. E' chiaro quindi che definire un oggetto "poligono irregolare" e derivare da esso come figli i poligoni regolari non è la strada migliore. Ci si trova a costruire per un caso estremamente generale con molta fatica campi e metodi che poi si devono via via abbandonare nei diversi oggetti figli, molto più regolari.

La gerarchia delle figure piane può invece partire da un oggetto di cui si dà un angolo e un lato: con queste informazioni si possono definire i metodi che costruiscono tutti i poligoni regolari, come triangolo equilatero e quadrato, ma anche pentagono, esagono, e tutti i poligoni di ordine superiore. Se si specificano due lati e un solo angolo, si può ricavare un rettangolo o un rombo. Se invece si specificano due angoli e un lato, si ricavano tutti i triangoli irregolari, e così via.

In questo modo ci si è spostati da un caso particolare (i poligoni regolari), che in virtù della sua regolarità intrinseca richiede poca informazione per

essere completamente definito, a casi più generali (i triangoli irregolari) per i quali è necessario specificare più informazioni e definire metodi più complessi.

Il processo di utilizzo dell'oggetto progenitore (poligono regolare) per ottenere i figli è un processo additivo: si aggiungono i campi via via che si rendono necessari per contenere nuove informazioni, si aggiungono metodi per realizzare prestazioni non presenti nell'oggetto originario.

ANCORA DEFINIZIONI

La programmazione orientata agli oggetti costituisce un metodo di programmazione che imita il normale modo di agire, ed è una naturale evoluzione delle prime innovazioni al progetto dei linguaggi di programmazione: è maggiormente strutturata rispetto ai precedenti tentativi di programmazione strutturata;

ed è maggiormante modulare e astratta rispetto ai precedenti tentativi di astrazione dei dati e occultamento dei dettagli. Un linguaggio di programmazione orientato, agli oggetti è caratterizzato da tre proprietà principali:

· *Incapsulamento: la possibilità di combinare un record con le procedure*

e le funzioni che lo manipolano per formare un nuovo tipo di dati: un

oggetto.

. *Ereditarietà: la possibilità di definire un oggetto e quindi utilizzarlo per*

costruire una.gerarchia discendente di oggetti, in cui ogni discendente

eredita l'accesso al codice e ai dati di tutti i suoi antenati.

· *Polimofismo: la possibità di assegnare ad un'azione un nome che sia*

condiviso lungo tutta la gerarchia degli oggetti; ed in cui ciascun

oggetto della gerarchia realizza l'azione secondo una sua modalità

specifica.

Le estensioni del linguaggio C++ vi permettono di sfruttare tutta

la potenza della programmazione orientata agli oggetti: maggior strutturazione e modularità maggiore astrazione; e possibilità di riutilizzo incorporata nel linguaggio stesso. La somma di queste caratteristiche da luogo a un codice che è, allo stesso tempo, più strutturato, estendibile e di facile manutenzione.

La sfida lanciata dalla programmazione orientata agli oggetti (00P) consiste nel fatto che a volte vi costringe a mettere da parte abitudini e processi mentali riguardanti la programmazione che sono ormai acquisiti da molti anni. Tuttavia, una volta superato questo scoglio iniziale, la 00P costituisce un metodo semplice, diretto e di livello superiore per risolvere numerosi problemi che affliggono i programmi software tradizionali.