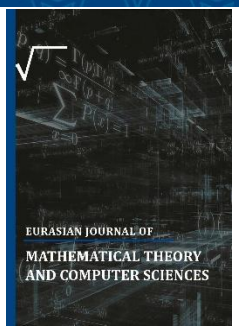
EURASIAN JOURNAL OF MATHEMATICAL THEORY AND COMPUTER SCIENCES
Innovative Academy Research Support Center
IF = 9.206 www.in-academy.uz/index.php/ejmtcs



USING THE GRAPHICAL TOOLS OF BORLAND C++ BUILDER

Botirov Muzaffarjon Mansurovich
Qoqon University, Department of Digital Technologies and Mathematics mbotirov@kokanduni.uz | botirovmuzaffarmansurov@gmail.com https://orcid.org/0000-0002-2078-1698
https://doi.org/10.5281/zenodo.18160692

ARTICLE INFO

Received: 29th December 2025
Accepted: 05th January 2026
Online: 06th January 2026

KEYWORDS

TCanvas, TFont, TPen, TBrush classes; TFont class properties: Color, Name, Size, Style; TPen class properties: Color, Mode, Width, Style; TBrush class properties: Bitmap, Color, Style.

ABSTRACT

Curious students who begin learning modern programming languages with the, regardless of the language, inevitably become acquainted with graphics libraries. The following article provides information on the graphical tools of Borland C++ Builder, as well as tips and instructions for creating multi-form applications.

Introduction. Main part. In the world of programming, there are many programming languages. However, apart from some minor differences, the foundation of all languages is the same. For example, almost every programming language has numbers, strings, arrays, and object-based data structures, as well as basic branching (if, else, switch) and iterative (loops: for, while) statements.

In fact, the most essential thing in programming is the algorithm. Problemsolving skills help you understand algorithms and come up with new ones. You can know the syntax of a programming language perfectly, but without algorithms you can't write an independent program.

The C++ Builder application encapsulates Windows GDI functions at various levels. In this case, one method is important, through which the graphic components render their images on the monitor screen. When a GDI function is called directly, you must pass the device context handle to these graphic components. This handle exposes the drawing objects you have selected—pens, brushes, fonts. After you have finished working with the graphics, you must restore the device context to its original state, and only then can you release it.

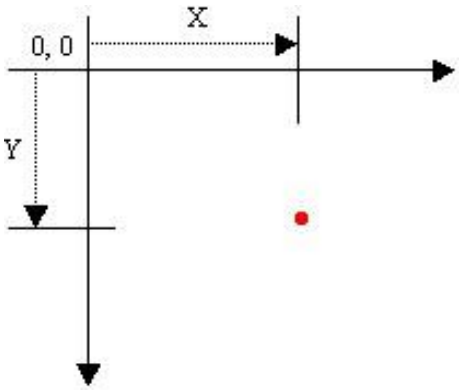
Instead of forcing you to work with graphics at that level of detail, C++ Builder offers a simple and complete interface for graphic components via their Canvas property. This property initializes the correct device context and releases it when you finish drawing. The BasePen has properties that operate under the names of the brush, pen, and font definitions.

The only thing a user needs to do when working with graphics components is specify the attributes of the drawing objects being used. You don't have to monitor system

resources when creating, selecting, and releasing objects. The base class itself takes care of this.

The purpose of this article is to provide information on developing applications in the Borland C++ Builder graphical environment.

In Microsoft Windows, when a drawing is created, the coordinates of the drawing area are based on the screen's top-left corner. Each point on the screen has its own reference at that point. This point is plotted in the Cartesian coordinate system starting from the origin (0,0), with the horizontal axis shifting to the right from (0,0) and the vertical axis shifting down from (0,0).



This default origin is simply the operating system's standard coordinate system. So if you draw a shape with **Canvas->Ellipse(-100, -100, 100, 100)**, you'll get an ellipse whose center is at the top-left corner of the screen. In this case, only the bottom-right 3/4 of the circle is visible.

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
Canvas->Pen->Color = clBlue;
Canvas->Ellipse(-100, -100, 100, 100);
}
```

In the same way, you can draw any geometric or non-geometric figure by using one of the TCanvas methods or create your own custom functions. For example, the following code draws vertical and horizontal lines that intersect in the middle of the form:

```
//---------------------------------------------------------------------
void __fastcall TForm1:: Button1Click(TObject *Sender)
{
   Canvas->Pen->Color = clBlack;
   Canvas->MoveTo(ClientWidth/2, 0);
   Canvas->LineTo(ClientWidth/2, ClientHeight);
   Canvas->MoveTo (0, ClientHeight/2);
   Canvas->LineTo(ClientWidth, ClientHeight/2);
}
//---------------------------------------------------------------------
```

Changing the default coordinate system in VCL

In some cases, you may not want to use the standard coordinate system for drawing. This is sometimes necessary for graphics, charts, and other related tasks. When using

Borland C++ Builder (or Delphi), there are two ways to change the origin of your drawing area's coordinate system.

You can use the MoveWindowOrg() function to move the (0,0) origin of your drawn area to any point (x,y). Its syntax is:

void __fastcall MoveWindowOrg(HDC DC, int DX, int DY);

The **MoveWindowOrg()** function takes three arguments. The first, the handle, must be a Win32 **HDC** handle for the device context. For a shape, paintbox, etc., this **HDC** is the handle to the canvas of the object you are drawing. The second argument, *DX*, is the starting point of the new horizontal axis. The final argument, *DY,* represents the starting point of the vertical axis.

After you call the **MoveWindowOrg()** function, you don't need to reference the device context handle, because the compiler will know which handle you're using and you can continue drawing the form.

After you call the **MoveWindowOrg()** function, it will refer to the device context handle for any drawing coordinates you perform under its call.

```
//---------------------------------------------------------------------
void __fastcall TForm1:: Button1Click(TObject *Sender)
{
HDC hDC = this->Canvas->Handle;
MoveWindowOrg(hDC, ClientWidth/2, ClientHeight/2);
Canvas->Pen->Color = clBlue;
Canvas->Ellipse(-100, -100, 100, 100);
Canvas->Pen->Color = clBlack;
Canvas->MoveTo(ClientWidth/2, 0);
Canvas->LineTo(ClientWidth/2, ClientHeight);
Canvas->MoveTo(0, ClientHeight/2);
Canvas->LineTo(ClientWidth, ClientHeight/2);
}
//---------------------------------------------------------------------
```

As you can see, using this function is very easy. In addition, you can see that our lines have disappeared (usually, you can still see the vertical line on the right side/border of the shape). In reality, as the circle has shifted, the lines have also shifted based on the current origin of the coordinate system. As mentioned above, as long as you keep track of where your coordinate system is, you can draw shapes however you want. **In** this **example,** we can draw shapes toward the center of the canvas.

In the example above, the origin was moved relative to the shape's client dimensions. Additionally, you can use a constant value to set the current origin. In fact, this can sometimes be faster and safer. In the new window, we place the form at the origin (400, 300). We can also shift the lines of our coordinate system:

```
//---------------------------------------------------------------------
void __fastcall TForm1:: Button1Click(TObject *Sender)
{
// Get the form's Handle to the Canvas
```

```
    HDC hDC = Canvas->Handle;
    // Change the origin of the coordinate system
    MoveWindowOrg(hDC, 300, 200);
    // Create a blue pen
    Canvas->Pen->Color = clBlue;
   //Draw a blue circle
    Canvas->Ellipse(-100, -100, 100, 100);
    //Change the    pen color to black
    Canvas->Pen->Color = clBlack;
    // Draw the vertical axis
    Canvas->MoveTo(0, 300);
    Canvas->LineTo(0, -300);
    // Draw the horizontal axis
    Canvas->MoveTo(-400, 0);
    Canvas->LineTo(400, 0);
}
//---------------------------------------------------------------------
```

The Win32 library provides another mechanism for changing the coordinate system used for drawing. The first thing you need to do is get the device context handle to the canvas of the object you are using as the drawing area. In short, you get this handle just as we did above.

A frequently used function similar to the above **MoveWindowOrg()** function is the **SetViewportOrgEx()** function. Its syntax is:

BOOL SetViewportOrgEx(HDC hdc, int X, int Y, LPPOINT lpPoint);

The first argument of this function is a handle to the device context you are using for drawing. The X and Y arguments specify the point that will serve as the new origin for the coordinate system. The last argument is a pointer to the origin of the previous point. Using **Win32's SetViewportOrgEx()** function is as easy as using the VCL's **MoveWindowOrg()** function. **Example**:

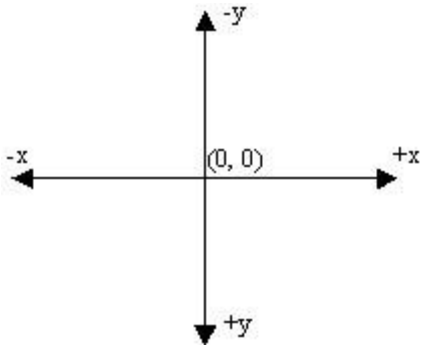
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   SetViewportOrgEx(Canvas->Handle, 300, 200, NULL);
   Canvas->Pen->Color = clRed;
   Canvas->Ellipse(-100, -100, 100, 100);
   Canvas->Pen->Color = clNavy;
   Canvas->MoveTo (0, 0);
   Canvas->LineTo(120, 0);
}
```

Consider the following example to try this out:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
     SetViewportOrgEx(Canvas->Handle, 300, 200, NULL);
   Canvas->Pen->Color = clRed;
```

```
    Canvas->Ellipse(-100, -100, 100, 100);
    Canvas->Pen->Color = clNavy;
    Canvas->MoveTo(0, 0);
    Canvas->LineTo(0, 120);
    Canvas->Pen->Color = clBlue;
    // Horizontal axis, from -X to +X
    Canvas->MoveTo(-300, 0);
    Canvas->LineTo(300, 0);
  // Vertical axis, from -Y to +Y
    Canvas->MoveTo(0, -200);
    Canvas->LineTo(0, 200);
    }
```

The SetViewportOrgEx() function changes the origin of the coordinate system. At the same time, it loads the new orientation of the axes. The horizontal axis moves positively to the right from the origin (0, 0). The vertical axis moves positively down from the origin (0, 0). This is illustrated as follows:



To experiment with this new direction, consider drawing a line at a 45º angle from the origin. Such a line is designated as positive down and positive right. In other words, the line is drawn from (0, 0) to (150, 150).

Example:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
SetViewportOrgEx(Canvas->Handle, 300, 200, NULL);
    Canvas->Pen->Color = clRed;
    Canvas->Ellipse(-100, -100, 100, 100);
    Canvas->Pen->Color = clBlue;
    Canvas->MoveTo(-300, 0);
    Canvas->LineTo(300, 0);
    Canvas->MoveTo(0, -200);
    Canvas->LineTo(0, 200);
    Canvas->Pen->Color = clFuchsia;
    Canvas->MoveTo (0, 0);
    Canvas->LineTo(150, 150);
}
```

```
//---------------------------------------------------------------------------
```

Microsoft Windows provides various options for controlling the orientation of the coordinate system you want to use for your application. In addition, it supports the unit system you want to use. We use the SetMapMode**()** function to control the coordinate direction and/or specify the unit system to use. Its syntax is:

int SetMapMode (HDC hdc, int fnMapMode);

The first argument of the function, as mentioned above, is the handle. The second argument is the types of unit systems for use.

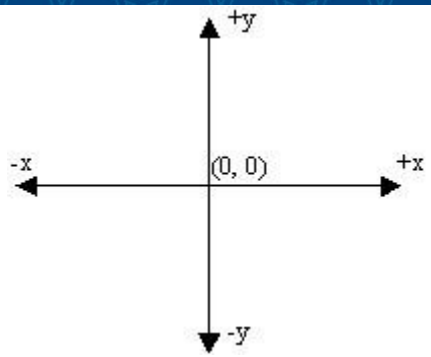
Available system units: **MM_TEXT**, **MM_LOMETRIC**, **MM_HIENGLISH**, **MM_ANISOTROPIC**, **MM_HIMETRIC**, **MM_ISOTROPIC**, **MM_LOMETRIC,** and **MM_TWIPS**.

The default standard map mode is MM_TEXT. In other words, unless you specify otherwise, this is the mode your application uses. In this map mode, it picks up and saves the dimensions you specify in your functions. Additionally, the axes are oriented so that they shift to the right from the horizontal axis (0, 0) and down from the vertical axis (0, 0). For example, the **OnPaint** event above can be rewritten as follows and produce the same result:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
HDC hDC = this->Canvas->Handle; SetMapMode(hDC, MM_TEXT);
   SetViewportOrgEx(hDC, 300, 200, NULL);
   Canvas->Pen->Color = clRed;
   Canvas->Ellipse(-100, -100, 100, 100);
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo (-300, 0);
   Canvas->LineTo(300, 0);
   Canvas->MoveTo(0, -200);
   Canvas->LineTo(0, 200);
   Canvas->Pen->Color = clFuchsia;
   Canvas->MoveTo (0, 0);
   Canvas->LineTo(150, 150);
}
```

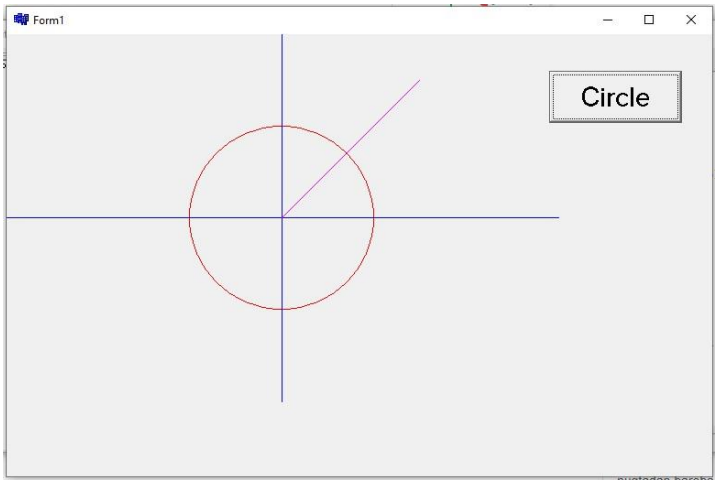
MM_LOENGLISH, like other map modes (except for the MM_TEXT shown above**)**, performs **two** operations. It reverses the direction of the vertical axis: the yaxis shifts upward from (0, 0):

EURASIAN JOURNAL OF MATHEMATICAL THEORY AND COMPUTER SCIENCES
Innovative Academy Research Support Center
IF = 9.206 www.in-academy.uz/index.php/ejmtcs



Also, each unit of measurement is multiplied by 0.01 inches, meaning the units are scaled down from the specified measurements. Consider the effect of the **MM_LOENGLISH** map mode on the above **OnPaint** event.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 HDC hDC = this->Canvas->Handle; SetMapMode(hDC, MM_LOENGLISH);
   SetViewportOrgEx(hDC, 300, 200, NULL);
   Canvas->Pen->Color = clRed;
   Canvas->Ellipse(-100, -100, 100, 100);
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo(-300, 0);
   Canvas->LineTo(300, 0);
   Canvas->MoveTo(0, -200);
   Canvas->LineTo(0, 200);
   Canvas->Pen->Color = clFuchsia;
   Canvas->MoveTo (0, 0);
   Canvas->LineTo(150, 150);
}
```



As you can see, the lines are now drawn with the positive and negative directions of the axes taken into account, which fulfills the requirements of the Cartesian system. At the same time, the lengths we used have been shortened, the circle is smaller, and the lines are shorter.

EURASIAN JOURNAL OF MATHEMATICAL THEORY AND COMPUTER SCIENCES
Innovative Academy Research Support Center
IF = 9.206 www.in-academy.uz/index.php/ejmtcs

Like the **MM_LOENGLISH** map mode, **MM_HIENGLISH** also corrects the orientation (actually, I shouldn't use the word "corrects," since I can assume MM_TEXT is an **anomaly**; (the map modes are provided so you can choose the direction you want at) shifts upward from the vertical axis (0, 0). Besides **MM_LOENGLISH**, the **MM_HIENGLISH** map mode scales each unit down by

0.001 inches, which can be significant and change the drawing appearance. Here is its effect: *void __fastcall TForm1::Button10Click(TObject *Sender)*

```
{
 HDC hDC = this->Canvas->Handle; SetMapMode(hDC, MM_HIENGLISH);
   SetViewportOrgEx(hDC, 300, 200, NULL);
   Canvas->Pen->Color = clRed;
   Canvas->Ellipse(-100, -100, 100, 100);
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo (-300, 0);
   Canvas->LineTo(300, 0);
   Canvas->MoveTo(0, -200);
   Canvas->LineTo(0, 200);
   Canvas->Pen->Color = clFuchsia;
   Canvas->MoveTo (0, 0);
   Canvas->LineTo(150, 150);
}
```

Notice that we are still using the same dimensions for the lines and the circle.

The MM_LOMETRIC map mode uses the same axis orientation as the previous two modes**.** On the other hand, **MM_LOMETRIC** multiplies each unit by 0.1 millimeters. This means that each unit is reduced by 10%. **Example**:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   HDC hDC = this->Canvas->Handle; SetMapMode(hDC, MM_LOMETRIC);
SetViewportOrgEx(hDC, 300, 200, NULL);
   Canvas->Pen->Color = clRed;
   Canvas->Ellipse(-100, -100, 100, 100);
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo (-300, 0);
   Canvas->LineTo(300, 0);
   Canvas->MoveTo(0, -200);
   Canvas->LineTo(0, 200);
   Canvas->Pen->Color = clFuchsia;
   Canvas->MoveTo (0, 0);
   Canvas->LineTo(150, 150);
}
```

The MM_HIMETRIC map mode uses the same orientation as the three modes above**.** Its units are obtained by multiplying each of the given units by 0.01 millimeters. **Example**:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
   HDC hDC = this->Canvas->Handle;
   SetMapMode(hDC, MM_HIMETRIC);
   SetViewportOrgEx(hDC, 300, 200, NULL);
   Canvas->Pen->Color = clRed;
   Canvas->Ellipse(-100, -100, 100, 100);
   Canvas->Pen->Color = clBlue;
   Canvas->MoveTo(-300, 0);
   Canvas->LineTo(300, 0);
   Canvas->MoveTo(0, -200);
   Canvas->LineTo(0, 200);
   Canvas->Pen->Color = clFuchsia;
   Canvas->MoveTo(0, 0);
   Canvas->LineTo(150, 150);
}
```

Matching units and coordinate systems

The map modes we have used so far have allowed us to select the orientation of the axes, especially the y-axis. Furthermore, we have been unable to apply any unit conversion to the dimensions shown in our drawings. This is because each of these mapping modes (MM_TEXT,

MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, and

MM_TWIPS) have hard-coded attributes, such as the orientation of their axes and how they convert the dimensions passed to them. What if you want to control the direction **and/or** conversion of the axes used in the dimensions you draw (have you ever used AutoCAD?).

Consider the following scenario. It is drawing a 200x200 pixel square with a red border and an aqua background. The square starts at (100, 100) on the negative side of both axes and continues to (100, 100) on the positive side of both axes. For a better illustration, the event also draws a 45º diagonal line starting from the origin (0, 0):

```
//---------------------------------------------- ---------------------------------- void __fastcall
TForm1::Button2Click(TObject *Sender){
   // Draw a red-bordered square. aqua background
   Canvas->Pen->Color = clRed;
   Canvas->Brush->Color = clAqua;
   Canvas->Rectangle(-100, -100, 100, 100);
   Canvas->Pen->Color = clBlue;
   // Start from the origin for a 45-degree diagonal line (0, 0)
   Canvas->MoveTo(0, 0);
   Canvas->LineTo (200, 200);
} 0
//---------------------------------------------- ---------------------------------- we will take only the
```

bottom right 3/4 of the square.

Imagine you want the origin (0, 0) to be placed in the middle of the shape, or more precisely, to place the origin at (340, 220)**.** The solution is to use the SetViewportOrgEx() function**. Example**

(we are not showing map mode, since MM_TEXT can be used as the standard for us)

```
//---------------------------------------------- ---------------------------------- void __fastcall TForm1::FormPaint(TObject *Sender)
{ HDC hDC = Canvas->Handle;  // set the form to Height = 340 and Width = 220
SetViewportOrgEx(hDC, 340, 220, NULL);
  // Draw a red border and an aqua-filled square
  Canvas->Pen->Color = clRed;
  Canvas-> Brush-> Color = clAqua;
  Canvas->Rectangle(-100, -100, 100, 100);
  Canvas->Pen->Color = clBlue;
  // Start from the origin (0, 0) for a 45-degree diagonal line
  Canvas->MoveTo(0, 0);
  Canvas->LineTo (200, 200);
}
//---------------------------------------------- ----------------------------------
```

Use the MM_ISOTROPIC or MM_ANISOTROPIC map modes to control the change of your unit system, the direction of the axes, and the units used in your application. The first thing you need to do is call the SetMapMode() function and specify one of these two map modes.

Example:

```
//---------------------------------------------- ---------------------------------- void __fastcall TForm1::FormPaint(TObject *Sender)
{
  HDC hDC = Canvas->Handle;
  SetMapMode(hDC, MM_ISOTROPIC);
  SetViewportOrgEx(hDC, 340, 220, NULL);
  // Draw a red bordered square with an aqua background
  Canvas->Pen->Color = clRed;
  Canvas->Brush->Color = clAqua;
  Canvas->Rectangle (-100, -100, 100, 100);
  Canvas->Pen->Color = clBlue;
  // Start from the origin (0, 0) for a 45-degree diagonal line
  Canvas->MoveTo(0, 0);
  Canvas->LineTo (200, 200);
}
//---------------------------------------------- ----------------------------------
```

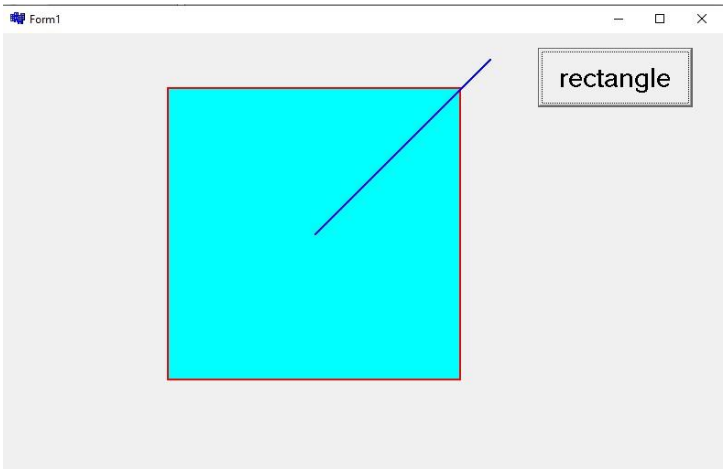
As a result of the above program, after you call the SetMapMode() function with MM_ISOTROPIC (or MM_ANISOTROPIC) as an argument, you don't have to stop there. The purpose of these two map modes is to allow you to control the orientation of the axes and the unit conversions. Therefore, after calling SetMapMode() and specifying

MM_ISOTROPIC (or MM_ANISOTROPIC), you must call SetWindowExtEx(). This function specifies how much each new unit is multiplied by to get the old or standard unit value. The syntax for the **SetWindowExtEx()** function is:

BOOL SetWindowExtEx(HDC hdc, int nXExtent, int nYExtent, LPSIZE lpSize);

Once again, the first argument of this function is the handle mentioned above. The second argument, nXExtent, specifies the maximum logical value of the horizontal axis. The third argument, nYExtent, specifies the maximum logical value of the vertical axis . The last argument, lpSize, is a pointer to the SIZE structure that stores the previous x and y values. If this argument is NULL, the argument is ignored. **Example**:

```
//---------------------------------------------- ---------------------------------- void __fastcall TForm1::Button14Click(TObject *Sender
{
  HDC hDC = Canvas->Handle;
  SetMapMode(hDC, MM_ISOTROPIC);
  SetViewportOrgEx(hDC, 340, 220, NULL);
SetWindowExtEx(hDC, 480, 480, NULL);
  // Draw a square with a red border and aqua fill
  Canvas->Pen->Color = clRed;
  Canvas-> Brush-> Color = clAqua;
  Canvas->Rectangle(-100, -100, 100, 100);
  Canvas->Pen->Color = clBlue;
  //45-degree diagonal line starting from the origin (0, 0)
  Canvas->MoveTo(0, 0);
  Canvas->LineTo(120, 120);
}
```



References:

1. Stanley Lippman. The C++ Language. Basic Course. Williams - M.: 2014.
2. Siddhartha Rao. Master C++ on Your Own in 21 Days. Williams - Moscow: 2013. 3. Nikita Kulytin. Microsoft Visual C++ in Problems and Examples. BHV St. Petersburg - St. Petersburg.: 2010.

3. B. Stroustrup. The C++ Programming Language. Special Edition. - M.: OOO "Bionom-Press", 2006. - 1104 pp.
4. Pavlovskaya T.A. C++: High-Level Language Programming – St. Petersburg: Piter. 2005. - 461 pp.
5. Sh. F. Madraximov, C.M. Gaynazarov. Fundamentals of Programming in C++. T. 2009.
6. FunctionX Tutorials (https://www.functionx.com)