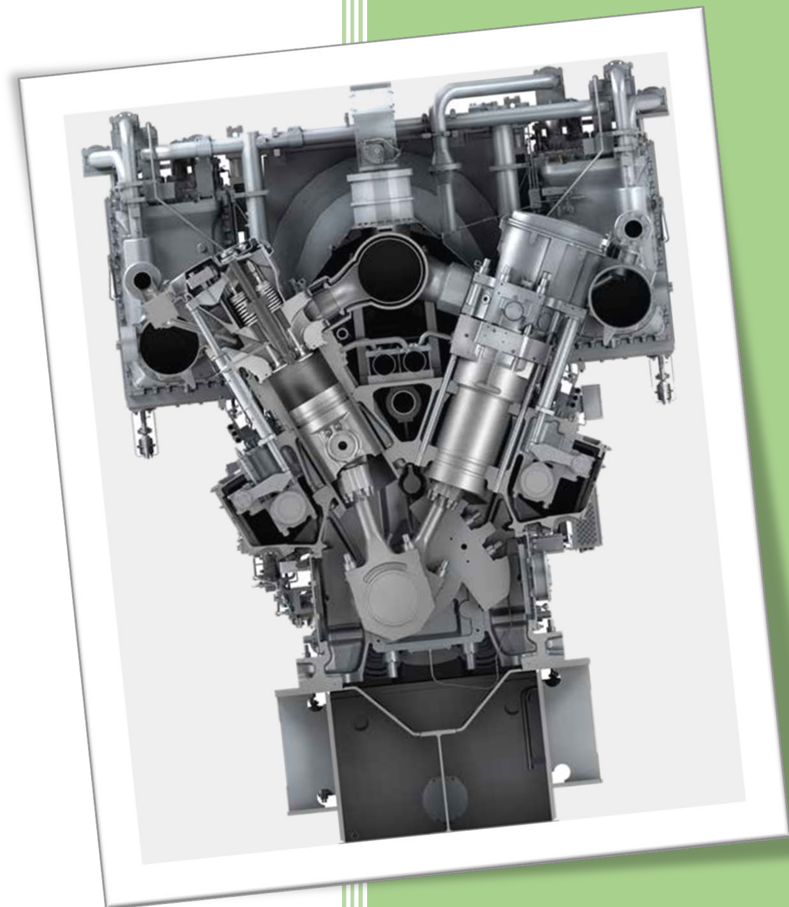
Dipl.-Phys.-Ing. Ralf Wirtz

Issue D2

2021

Programming SimplexNumerica

with AngelScript



V18

Programming & Visualization

Programming SimplexNumerica with AngelScript

This documentation is provided to familiarize you with the fundamentals of the *SimplexNumerica* programming examples *with **AngelScript,*** to find in the setup folder *Scriptings*.

AngelScript is a scripting language with a syntax that is very similar to C++. It is a strictly typed language with many of the types being the same as in C++. This part of the documentation will explain some concepts of how to use *AngelScript* in general, but a basic knowledge of the language will be needed to understand all of the concepts.

Please have a look to the *AngelScript* web page at www.AngelCode.com/AngelScript/

→ ***AngelScript*** is made by ***Andreas Jönsson***

→ ***SimplexNumerica*** and its programming interface to *AngelScript* is made by ***Ralf Wirtz***

```
Info
This manual is an extension to the main SimplexNumerica manual.
```

```
Info
When SimplexNumerica starts the first time, then it copies the folder
Scriptings to your user directory, so that you can manipulate the code.
Next time, only if the Scriptings folder is not available or new scripts
are available, then it copies again!
```

Please visit the last page for the license agreement!

SIMPLEXNUMERICA, *SIMPLEXNUMERICA*, SIMPLEXEDITOR AND Simplexety ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL THE AUTHOR BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING INCIDENTAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF THE USE OF THE PROGRAM, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

What is new in this document?

We will start here with a table that references to modified or new chapters in this document and contingently new scripts in the folder <Scriptings>.

Each table below contents a new issue of this document. The actual issue can be found on the front page – the first page of this document

Issue D1-2017

	Chapter	News	Script
	2.25 Spreadsheet Base Functions	Base Spreadsheet Script Functions	`Spreadsheet Script.cpp`

	3 Call Script from Button	Click on a button and call a script to operate a double approach	`Hello World.sx`
			`Hello World.cpp`
			`Add.sx`
			`Calc.cpp`

Issue C4-2017

	Chapter	News	Script
	6.4 Strings	New chapter about strings	`Strings.cpp`
	2.23 Database Import	Database Import	`Database Import.cpp`
	2.24 WinCC Database Import	WinCC Archive Import	`WinCC Database Import.cpp`
	2.20 Import Excel Standard File	New function: `app.ImportExcelFile`	`ImportExcelFile.cpp`
	2.8 Export Graphic as Image	Shows how to export chart object(s) as bitmap or image	`Export Chart as Bitmap.cpp`

	2.22 Rotate 3D Surface Plot	New functions:
		`SetRotationAngle(rot)`
		`SetElevationAngle(elev)`
		`SetTwistAngle(twist)`
			`Animate Surface Plot.cpp`

	2.6 Check Graph	Code around the new function: `CheckGraph(…)`	`Check Graph.cpp`
	2.7 Remove Graph	Remove the current (active) graph	`Remove Current.cpp`
		Remove any other graph	`Remove Graph.cpp`
		Use `parseInt(…)` to select the right graph to remove	`Remove Greater than.cpp`
	2.4 Get Chart Object	To get an instance of the chart via script use `GetChart()`	`SelectChartEx.cpp`

	2.5 Select Active Graph	To activate a graph uset the new function `SetActiveGraph(i)` `i = 0,1,..n-1`	`SetActiveGraph.cpp`
		The same in a loop…	`SetActiveGraph V2.cpp`

Content

CONTENT 5

1 DEVELOPMENT 9

2 PROGRAMMING IN SIMPLEXNUMERICA 10

2.1 Default Script 11

2.2 Hello World 15

2.3 Make Chart 16

2.4 Get Chart Object 21

2.5 Select Active Graph 22

2.6 Check Graph 24

2.7 Remove Graph 26

2.8 Export Graphic as Image 30

2.9 Set Label 32

2.10 Arrange Charts 38

2.11 Set Property 42

2.12 Load Project 46

2.13 Import and Calc Data 47
2.13.1 Manipulate sample data and write it back to the chart memory 51

2.14 Make Text Label 54
2.14.1 Change Text Color 57
2.14.2 Change Font Name 58
2.14.3 Change Font Size 58
2.14.4 Change Font Style 58
2.14.5 Change Font Alignment 58
2.14.6 Change Font Justification 59
2.14.7 Change Font Opacity 59
2.14.8 Change Text itself 59
2.14.9 Move any Shape 59

2.15 Make Drawing Shape 62

2.16 Make Chart on Layer 64

2.17 Update Layer Window 65

2.18 Make Chart on Layer Extended 66

2.19 Write to Excel File 72

2.20 Import Excel Standard File 75

2.21 Make Surface Plot 77

2.22 Rotate 3D Surface Plot 80

2.23 Database Import 82
2.23.1 Make an instance of the database class 82
2.23.2 Connect to Database 82
2.23.3 Run Query 83
2.23.4 Save Query Results 83
2.23.5 Transfer to DataSheet 83
2.23.6 Release Interface 84

2.24 WinCC Database Import 86

2.25 Spreadsheet Base Functions 88

3 CALL SCRIPT FROM BUTTON 100

3.1 Make a shape to a text shape 100

3.2 Method 1: Script with a main() function 101

3.3 Method 2: Script with any C++ function 103
3.3.1 Approach I 103
3.3.2 Approach II 105

4 SIMPLEX REMOTE CONTROL (SIMPLEXIPC) 108

4.1 User Interface 109

4.2 Send an Example 111

5 IPC TEST CLIENT 112

5.1 Source Code 113

5.2 Usage 114

6 ANGELSCRIPT 115

6.1 Unary operators 117

6.2 Binary and ternary operators 117

6.3 Expressions 119
6.3.1 Assignments 119
6.3.2 Compound assignments 119
6.3.3 Function call 119

6.3.4 Type conversions 120
6.3.5 Math operators 120
6.3.6 Bitwise operators 121
6.3.7 Logic operators 121
6.3.8 Equality comparison operators 121
6.3.9 Relational comparison operators 122
6.3.10 Identity comparison operators 122
6.3.11 Increment operators 122
6.3.12 Indexing operator 122
6.3.13 Conditional expression 122
6.3.14 Member access 122
6.3.15 Handle-of 123
6.3.16 Parenthesis 123
6.3.17 Scope resolution 123

6.4 Strings 124
6.4.1 String object and functions 125
6.4.2 Methods 125
6.4.3 Functions 127

6.5 Template Arrays 128
6.5.1 Array object and functions 129

6.6 Data Types 130
6.6.1 void 130
6.6.2 bool 130
6.6.3 Integer numbers 130
6.6.4 Real numbers 131
6.6.5 Arrays 131
6.6.6 Objects 132
6.6.7 Object handles 132
6.6.8 Strings 132

6.7 Statements 134
6.7.1 Variable declarations 134
6.7.2 Expression statement 134
6.7.3 Conditions: if / if-else / switch-case 134
6.7.4 Loops: while / do-while / for 135
6.7.5 Loop control: break / continue 136
6.7.6 Return statement 136
6.7.7 Statement blocks 136

6.8 Property Assessors 137

6.9 Globals 138
6.9.1 Functions 138
6.9.2 Variables 138
6.9.3 Classes 139
6.9.4 Interfaces 139
6.9.5 Imports 140
6.9.6 Enums 140
6.9.7 Typedefs 140

6.9.8 Object Handles 142
6.9.9 Object life times 142

6.10 Script Classes 144

6.11 Operator overloads 146

7 END-USER LICENSE AGREEMENT 149

1 Development

The idea for *SimplexNumerica* sprung out of my own desires to create a relatively simple data plotter. Thus, *SimplexNumerica* started out as a small side project of mine in 1986. I have previously worked on other programs and something I noticed early on was the benefits of having a good base layer. In fact, a lot of my work with *Simplex* has revolved around building programs like *SimplexGraphics*, *Simplexety* and *SimplexEditor* as the base layer.

SimplexNumerica is designed to provide the power and functionality to satisfy the most demanding data plotting requirements. It can handle arrays up to the limits of virtual memory, and will work with 32 and 64-bit editions of Microsoft Windows™ like Windows 10.

SimplexNumerica has a wide-ranging library of 2D and 3D charts with a large section based on numerical mathematics like approximation and interpolation algorithms. Equipped with genuine object-oriented vector diagrams with context sensitive pull down menus and properties, also the report and layout windows facilitate the ease-of-use and operation of the program. Likewise, the chart module integrated into the user interface places its elements (lines, polygons, ellipses etc.) in an object-oriented manner. Icons and Ribbonbar menus for selecting, increasing, grouping etc. are also intuitively present. The diagram types and numeric functions can be checked in separate data sheets. The tool windows are dynamically updated to show the most important functions; mouse-clicks are the only action necessary for most operations.

Complex operational sequences are taken care of automatically as far as possible by the program. Auto-scale routines permit the highest automation. The interactive nature of data analysis limits your user-inputs to that which are only necessary. When just getting the job done is work enough, the last thing you need is to waste time having to learn yet another computer application. Your experience with other tools should be relevant to each new application, making it possible to sit down and use that new application right away. That is why *SimplexNumerica* is so popular. Whether you simply need a powerful extension for Excel, a tool for plotting row data, or whatever, *SimplexNumerica* does hopefully what you want and the way you would expect. *SimplexNumerica* is designed to provide the power and functionality to satisfy the most demanding plotting needs.

SimplexNumerica has been implemented according to the *Microsoft Windows Guidelines for Accessible Software Design*, so great attention has been paid to making it easy for both beginners and experienced users.

If you still have further questions, please do not hesitate to contact us.

- Ralf Wirtz, Software Engineer and Developer

Email: support@SimplexNumerica.com
Web: www.SimplexNumerica.com

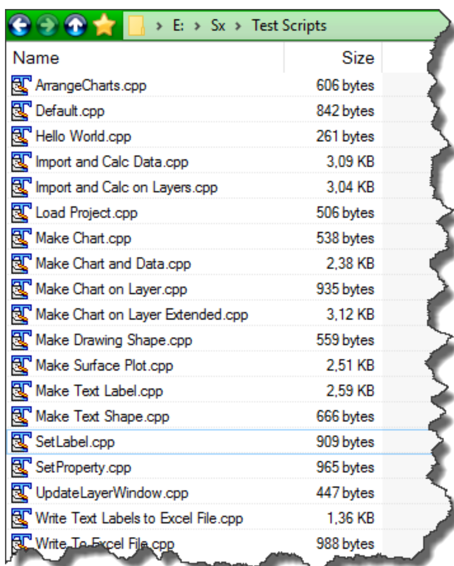
2 Programming in SimplexNumerica

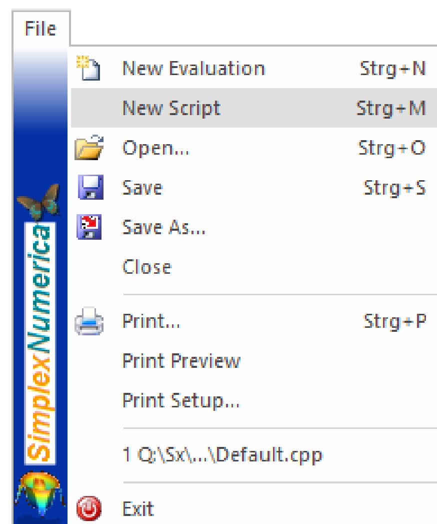
The scripting language inside *SimplexNumerica* is ***AngelScript***. ***AngelScript*** is a scripting language with a syntax that is very similar to C++. Please have a look to the ***AngelScript*** chapter 6, here in the ***SimplexNumerica*** programming manual.

This chapter will be a kind of tutorial for the inbuilt scripting host functionality in ***SimplexNumerica.*** All script examples in this chapter are copied from the setup folder `Scriptings`.

You can simply open the `Scriptings` folder with the help of the start-up dialog - please click on *Open Sample Script*.

Press the button ***Open Sample Script*** and you will get the Fileselectbox with the list of script files:





→ But we will simply start with the Ribbonbar Pulldownmenu ***File***, menu item ***New Script***.

The editor will be filled with a default script. You can load that script also from the scriptings folder <..\Scriptings\Default.cpp>

2.1 Default Script

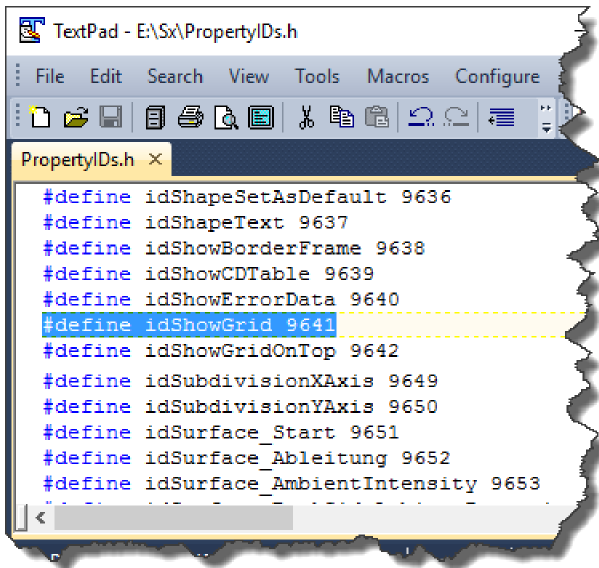
Here the whole script and underneath the explanation.

```
#pragma extension "corelib"

void main()
{
  Application app("My App");
  string strQuestion;
  string str = "Hello" + " World!";
  alert(str);

  bool ret = MyDummyFunction(str, strQuestion);
  if (ret)
    app.Output(alertYes("You said Ok.\n" + strQuestion));
  else
    app.Error(alertYes("You said No.\n" + strQuestion));
}

bool MyDummyFunction(string str, string& strMyQuestion)
{
  string str2 = str.replace("Hello", "Well, what shall I say to this", false);
  str2 = str2.replace("!", "?", false);

  if (alertOk(str2) == 1)
  {
    strMyQuestion = "That's right?";
    return true;
  }
  else
  {
    strMyQuestion = "Is that right?";
    return false;
  }
}
```

<u>Explanation:</u>

```
#pragma extension "corelib"
```

This is a preprocessor instruction of the inbuilt scripting host. Each program needs this in the first lines of code.



In addition, it loads the header file `<PropertyIDs.h>` that you can find in the root setup folder of ***SimplexNumerica***.

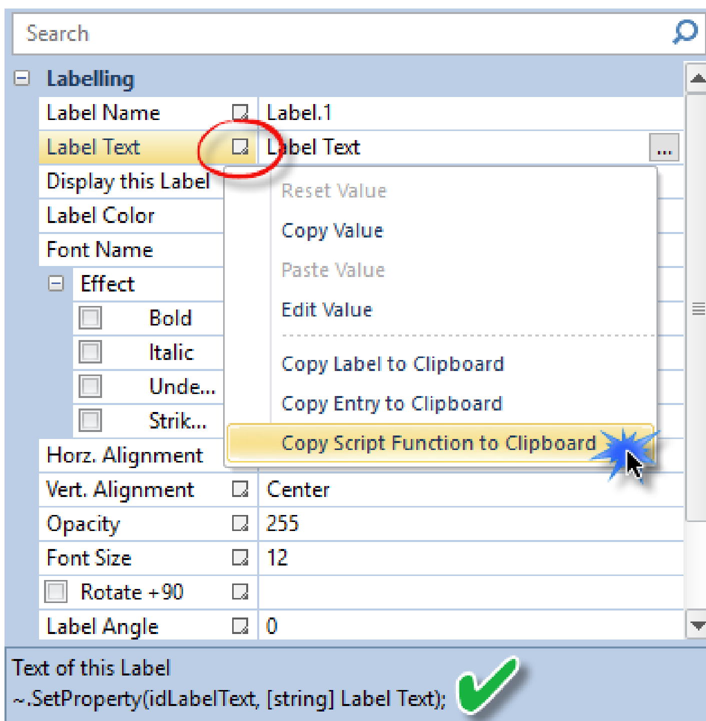
It defines each used property identifier (Id) of the ***Property Windows*** content, e.g. the chart properties.

These Ids can be used in corresponding script functions, like

```
//…
Chart ch = app.MakeChart(..);
ch.SetProperty(idShowGrid, false);
//…
```

→ Please draw a ***Text Label*** and open its properties.

When you click on a list entry, then you can see a short help note at the bottom and the property command for the scripting engine.



As you can see, the list entries on the first column have a small box on the right side. Click on this box to open the Popupmenu. You can simply read what you can do with the menu entries. As an example we will show you the use of the menu ***Copy Script Function to Clipboard***.

The picture above grabs the function `~.SetProperty(idLabelText, [string] Label text);`

The function will be used like:

```
Chart ch = app.MakeChart("My Chart", idChartTypePhysics, 100, 100, 400, 300);

ch.SetProperty(idLabelText, "Hello World");
```

```
Note
Use SetProperty() to set the properties (simple to use, but slow) or use
the individual object functions from the scripting host (see next
chapters).
```

```
void main()
{
  //..
}
```

Each C++ program has an entry point in form of a main function. As easy to apply, in *SimplexNumeria*, this function is called ***main()***.

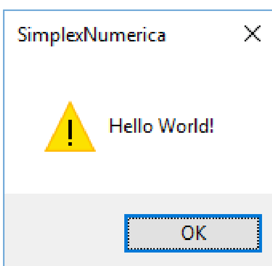
```
Note
Each script file needs to have exactly one main() function.
```

```
Application app("My App");
```

The root class of the scripting host is called Application, an instance (here the word `app`) can be called whatever name you like. The variable name `app` is short and clear to understand, nowadays. We can use it later in other functions. The string `"My App"` is the name of your application script program.

```
string strQuestion;
string str = "Hello" + " World!";
alert(str);
```

Declare a `string` variable and display it in a messagebox with the `alert` function.



→ Function `alert` displays a text string and an ***OK*** button.

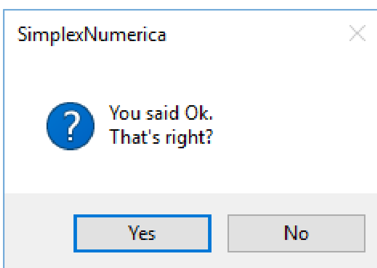
You can add strings already in the declaration line: `string str = "Hello" + " World!";`

```
bool ret = MyDummyFunction(str, strQuestion);
```

This is a typical C++ function call with one return value and two arguments. But as we can see later, the first argument is ***Call by Value*** and the second is ***Call by Reference*** (see literature). The return value is a flag, declared as Boolean. Both arguments are strings.

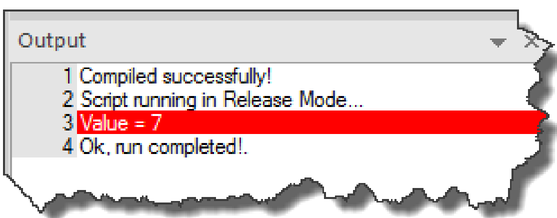
```
if (ret)
  app.Output(alertYes("You said Ok.\n" + strQuestion));
else
  app.Error(alertYes("You said No.\n" + strQuestion));
```

The ***if statement*** decides which answer the program gives to the user in form of a messagebox.



→ Function `alertYes` displays a messagebox with a text string and a ***Yes*** and ***No*** button.

The function `alertYes` is enclosed from another function ***Output(..)*** (if statement equal true) or ***Error(..)*** (if statement equal false).



Function ***Output*** shows the result of an original Windows Messagebox in the Output Window and function ***Error*** does the same, but highlighted in red.

→ If you have a closer look to the results of the code, then it should be apparent, that the result is an integer (Value = 7) instead of an Boolean value.

But, everything equal to zero is ***false*** else ***true***.

```
bool MyDummyFunction(string str, string& strMyQuestion)
```

As already told above, this is a typical C++ function with one Boolean return value and two string arguments. But, the first argument is ***Call by Value*** and the second is ***Call by Reference*** ***(Symbol &)***.

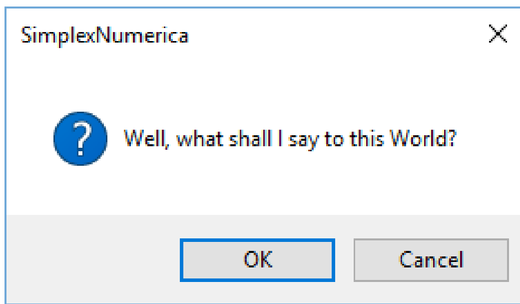
Call by Value means mainly for you: The function cannot manipulate your argument.

Call by Reference means mainly for you: The function can manipulate your argument.

→ To demonstrate this behavior, next, we will change something inside the function.

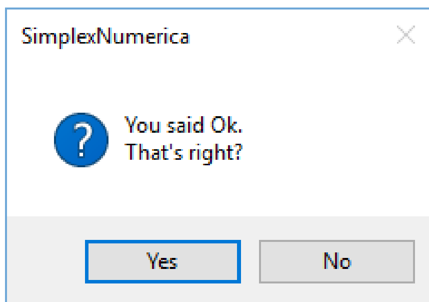
```
string str2 = str.replace("Hello", "Well, what shall I say to this", false);
str2 = str2.replace("!", "?", false);
```

We will use the function `replace(..)` to replace the word `"Hello"` against the text `"Well, what shall I say to this"`, so that we get the sentence `"Well, what shall I say to this World"`.



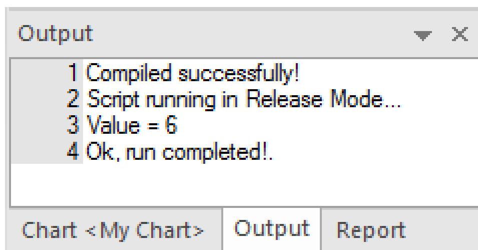

```
if (alertOk(str2) == 1)
{
  strMyQuestion = "That's right?";
  return true;
}
else
{
  strMyQuestion = "Is that right?";
  return false;
}
```

→ Function `alertOk` displays a messagebox with a text string and a ***OK*** and ***Cancel*** button. The ***if statement*** decides which text in the ***Call by Reference*** variable comes out of the function.



As already told above, the right text will be coming back and displayed to the left messagebox.

Please, have a look to the Output Window.



→ The best result is when everything works perfect…

That's it!

2.2 Hello World

Here that example, without we can go further: ***Hello World!*** How short that is in ***SimplexNumerica's AngelScript*** implementation - that should be very impressive…

```
#pragma extension "corelib"

void main()
{
  Application app("My App");
  string str = "Hello World!";
  alert(str);
}
```

→ Everything is explained before, in the previous chapter.

2.3 Make Chart

Objectives:

1. Make a chart
2. Set a chart property
3. Set the chart width
4. Select the chart

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  app.NewEval();

  Chart ch = app.MakeChart("My Chart", idChartTypePhysics, 100, 100, 400, 300);

  ch.SelectPropertyGroup("Chart Properties");
  ch.SetProperty(idChartName, "My Chart");

  ch.SetWidth(ch.GetWidth() * 2);

  app.SelectChart("My Chart");

  // Finally update properties on screen
  app.UpdateWindows();
}
```

```
app.NewEval();
```

Use this short member function of the ***Application*** class to make a new empty evaluation window.

Definition:

```
void NewEval()
```

```
Chart ch = app.MakeChart("My Chart", idChartTypePhysics, 100, 100, 400, 300);
```

The main class from the scripting host to make a chart is called MakeChart, an instance can be called whatever name you like. The variable name `ch` is short and succinct. We can use it later in other functions or to call its member functions. The string `"My Chart"` is the name of the chart.

Definition:

```
Chart MakeChart(string stdChartName, uint type, uint x1, uint y1, uint width, uint height)
```

See next page for the arguments…

<table>
<tr><th>Variable</th><th>Function</th></tr>
<tr><td><code>stdChartName</code></td><td>The name of the chart</td></tr>
<tr><td><code>Type</code></td><td>The type of the chart
Use one of the following Ids:
idChartTypeMath
idChartTypePhysics
idChartTypeSmith
idChartTypeTernary
idChartTypePie
idChartTypeXYLine
idChartTypeSurface
idChartTypePolarV2
idChartTypeBar
idChartTypeContourPlot
idChartTypeMeter
idChartTypeMisc
idChartTypeExLine
idChartTypeExPie
idChartTypeExPie3D
idChartTypeExPyramid
idChartTypeExPyramid3D
idChartTypeExFunnel
idChartTypeExFunnel3d
idChartTypeExVerticalBar
idChartTypeExHorizontalBar
idChartTypeExHistogram
idChartTypeExArea
idChartTypeExStock
idChartTypeExBubble
idChartTypeExLongdata
idChartTypeExHistoricalline
idChartTypeExPolar
idChartTypeExDoughnut
idChartTypeExDoughnut3D
idChartTypeExTorus3D
idChartTypeExTernary
idChartTypeExColumn3D
idChartTypeExBar3D
idChartTypeExLine3D
idChartTypeExArea3D
idChartTypeExSurface3D
idChartTypeExDoughnutNested
idChartTypeExCombined3DChart
idChartTypeExCombined2DChart
Info: Not all are implemented, yet!</td></tr>
</table>

Next Arg.	Sets the position and size of the chart to the specified values.
x1	The x coordinate of the top left corner.
y1	The y coordinate of the top left corner.
width	The width of the chart.
height	The height of the chart.

Return Value:

Returns a chart instance. An object variable, that can be used in front of the member functions.

```
ch.SelectPropertyGroup("Chart Properties");
```

This member function of the ***Chart*** class selects the ***Property Window Group*** by name. See left combobox and the equal name of the argument.

```
ch.SelectPropertyGroup("Chart Properties");
```

Definition:

```
void SelectPropertyGroup(string group)
```

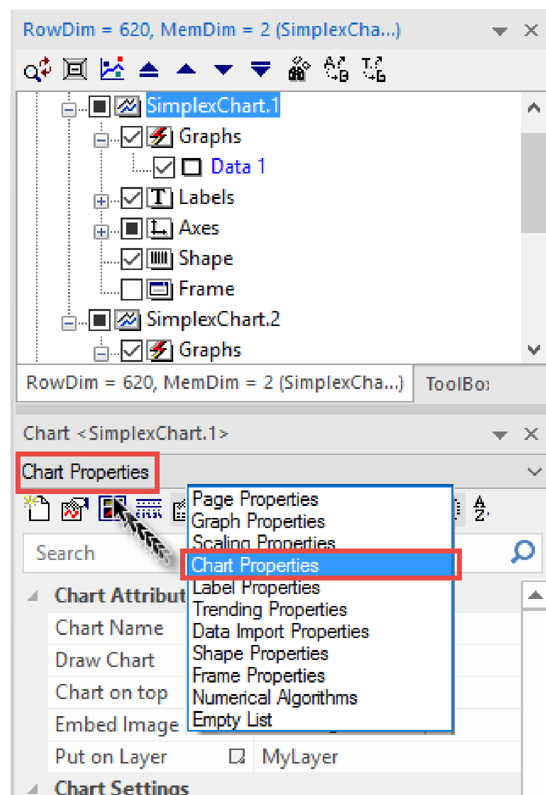
Return Value:

void means nothing

Arguments:

Combobox entry name

Chart Properties:



```
ch.SetProperty(idChartName, "My Chart");
```

This member function of the ***Chart*** class selects the property itself.

→ Please have a look to chapter 2.1, *Default Script* and then the part about the function ***SetProperty(..)*** !

Definition:

```
void SetProperty(uint Id, bool var)
```

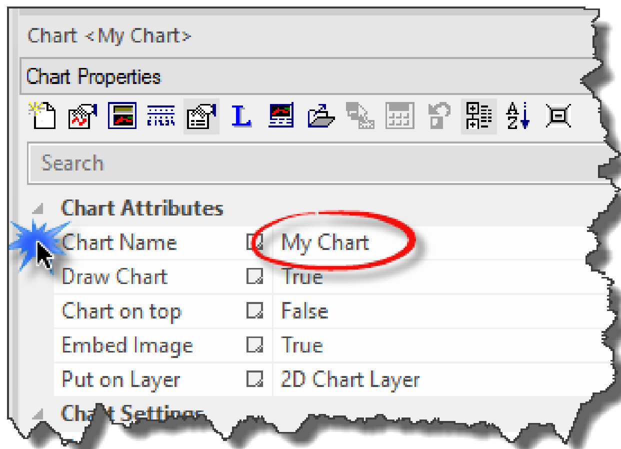
or overloaded:

```
void SetProperty(uint Id, string var)
```

Arguments:

Variable	Function
Id	Identifier from the file `<PropertyIDs.h>`
var	Value to set into the property cell.

Example:




```
ch.SetProperty(idChartName, "My Chart");
```

→ The entry ***Chart Name*** has changed to the new name “My Chart”.

or

```
ch.SetProperty(idDrawChart, false);
```

→ The entry ***Draw Chart*** will be changed to the ***false***. As a consequence, the chart contour will not be drawn.

```
ch.SetWidth(ch.GetWidth() * 2);
```

Instead to use the function `SetProperty()`, you can use (if available) also a distinctive member function of the ***Chart*** class.

Definition:

```
void SetWidth(float width)
```

Arguments:

Variable	Function
width	Width of the chart.

Definition:

```
float GetWidth()
```

Arguments:

nothing

Return Value:

Width of the chart at the present moment.

```
app.SelectChart("My Chart");
```

Use this function to select a chart. Tell the program which chart you mean. If the program has already selected the chart, then it does nothing.

Definition:

```
void SelectChart(string name)
```

Return Value:

nothing

Arguments:

Variable	Function
Name	Name of the chart, that you want to select.

Definition:

```
void UpdateWindows()
```

Updates properties on screen. Primary the ***Chart Explorer***, ***Property Window*** and ***Layer Window***.

```
Hint
You can always put this function at the end of the main function!
void main()
{
  //..
  app.UpdateWindows();
}
```

That's it!

You can find this script file in your ***SimplexNumerica*** setup folder: `<..\Scriptings\Make Chart.cpp>`

2.4 Get Chart Object

Objectives:

1. Load any Evaluation
2. Get the chart object
3. Set a chart property
4. Select the chart

If a chart already exists with a certain chart-name as part of the evaluation, then you can get this chart object:

Definition:

```
Chart GetChart()
```

Return Value:

Chart Object, like `Chart ch = app.GetChart("MyChart");`

Arguments:

nothing

Here a sample script for getting a chart object:

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  app.LoadEval("e:\\test.sx");

  // Chart Object
  Chart ch = app.GetChart("MyChart");

  ch.SelectPropertyGroup("Chart Properties");
  ch.SetProperty(idChartName, "My New_Chart-Name");

  app.SelectChart("MyChart");

  // Finally update properties on screen
  app.UpdateWindows();
}
```

(from ..\Scriptings\SelectChartEx.cpp)

2.5 Select Active Graph

Objectives:

1. Load any Evaluation
2. Get the chart object
3. Set Active Graph
4. Redraw chart

To select one of the chart's graph (= marker or/and curve) use the following function:

Definition:

```
void SetActiveGraph(int graphNo); graphNo from 0 … n-1
```

Return Value:

nothing

Arguments:

Graph Number (from 0 to n-1)

```
Info
If you set graphNo to high, then the program sets the last graph active!
```

Here a simple sample script for setting the active graph:

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  const string filename = "E:\test.sx";

  app.Output(filename);

  if (app.FileExist(filename))
  {
    app.LoadEval(filename);

    Chart ch = app.GetChart("My Chart");

    ch.SetActiveGraph(0); // 0 is the first graph

    app.Redraw();
  }
  else
  {
    app.Error("File does not exist!");
  }
}
```

(from ..\Scriptings\SetActiveGraph.cpp)

Next example script endless loops through a number of graphs:

```
#pragma extension "corelib"

#define ever (;;)
#define NUMBER_OF_GRAPHS 10

void main()
{
  Application app("Simple App");

  string filename = "c:\\Test.sx";

  if (app.FileExist(filename))
  {
    app.LoadEval(filename);

    Chart ch = app.GetChart("My Chart");

    int graph = 1;

    for ever
    {
      if (app.EscapeLoop()) // Press key Esc and leave the loop
        break;

      app.DelayMS(300);

      if (!app.IsGraphicsViewAvailable()) // Do NOT forget this!
        break;

      ch.SetActiveGraph(graph);

      if (graph++ >= NUMBER_OF_GRAPHS)
        graph = 1;
    }
  }
  else
  {
    app.Error("File does not exist!");
  }
}
```

(from ..\Scriptings\SetActiveGraph V2.cpp)

2.6 Check Graph

Objectives:

1. Load any Evaluation
2. Get the chart object
3. Uncheck all graphs
4. Check graph after graph in a loop
5. Export chart object as '*.png' image

To check/uncheck one of the chart's graph (marker or/and curve) use the following function:

Definition:

```
void CheckGraph(int graphNo, bool check = true); graphNo from 0 … n-1
```

Return Value:

nothing

Arguments:

Graph Number (from 0 to n-1)
check = true: checked; check = false: unchecked

To check/uncheck all of the chart's graphs use the following function:

Definition:

```
void CheckAllGraphs(bool check = true);
```

Return Value:

nothing

Arguments:

check = true: checked; check = false: unchecked

Next working code loops through a number of graphs, checks each after have unchecked all and exports each as an '*.png' image.

```
#pragma extension "corelib"

#define STRING_NOT_FOUND -1

void main()
{
  Application app("My App");

  const string filenameEval = "H:/test.sx";

  if (app.FileExist(filenameEval))
  {
    app.LoadEval(filenameEval);
```

```
      Chart ch = app.GetChart("Test Chart");

      bool abbruch = false;

      for (int j = 0; j <= 10; j++)
      {
        ch.CheckAllGraphs(false);

        string index;

        for (int i = 0; i < ch.GetNumberOfGraphs(); i++)
        {
          if (app.EscapeLoop()) // Press key Esc and leave the loop
          {
            abbruch = true;
            break;
          }

          string graphName = ch.GetGraphName(i);

          string sub = graphName.substr(2, 4); // e.g.: 0001

          int val = parseInt(sub);

          if (val == j)
          {
            app.Output("Check Graph: " + graphName);
            ch.CheckGraph(i, true);
            index = sub;
          }
        }

        if (abbruch)
          break;
        else
          app.DelayMS(500);

        string newFilename = "H:\\Images\\Image" + index + ".png";

        app.Output(newFilename);

        app.ExportChartObjectAsBitmap(true, newFilename);
      }

      app.UpdateWindows();
    }
    else
    {
      app.Error("File not found:" + filenameEval);
    }
}
```

(from ..\Scriptings\Check Graph.cpp)

2.7 Remove Graph

Objectives:

1. Load any Evaluation
2. Get the chart object
3. Loop through all graphs and remove any one

To remove one of the chart's graph (marker or/and curve) use the following function:

Definition:

```
void RemoveGraph(int graphNo); graphNo from 0 … n-1
```

Return Value:

nothing

Arguments:

Graph Number (from 0 to n-1)

Next example looks for a Graph with the name abc123 and removes this:

```
#pragma extension "corelib"

#define STRING_NOT_FOUND -1

void main()
{
  Application app("My App");

  const string filenameEval = "H:\\test.sx";

  if (app.FileExist(filenameEval))
  {
    app.LoadEval(filenameEval);

    Chart ch = app.GetChart("My Test Chart");

    for (int i = 0; i < ch.GetNumberOfGraphs(); i++)
    {
      if (app.EscapeLoop()) // Press key Esc and leave the loop
        break;

      string graphName = ch.GetGraphName(i);
      app.Output("Graph: " + graphName);

      if (graphName.findFirst("abc123") != STRING_NOT_FOUND)
      {
        app.Output("Remove Graph: " + graphName);
        ch.RemoveGraph(i);
      }
```

```
    }

    app.UpdateWindows();
  }
  else
  {
    app.Error("File not found:" + filenameEval);
  }
}
```

(from ..\Scriptings\Remove Current.cpp)

Next example adds x- and y-Values from Graph1 and puts the result in y-Value of Graph2 and then after removes the graph.

```
#pragma extension "corelib"

#define FIRST_GRAPH 0
#define SECOND_GRAPH 1

int iMin(int a, int b)
{
  return ((a < b) ? a : b);
}

int iMax(int a, int b)
{
  return ((a > b) ? a : b);
}

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetSimplexAppPath();

  string filename = simplexAppPath + "Examples\\MainPlots\\Calc via Script.sx";

  if (app.FileExist(filename))
  {
    app.LoadEval(filename);

    // New function
    Chart ch = app.GetChart("MyChart");

    int m = ch.GetNumberOfGraphs();
    app.Output(m);

    //
    // Example
    // Add x- and y-Value from Graph1 and put the result in y-Value of Graph2
    //
```

```
    int n1 = ch.GetNumberOfSampleData(FIRST_GRAPH);
    int n2 = ch.GetNumberOfSampleData(SECOND_GRAPH);

    int N = iMin(n1, n2);

    for (int row = 0; row < N; row++)
    {
      double x = ch.GetDataX(row, FIRST_GRAPH);
      double y = ch.GetDataY(row, FIRST_GRAPH);

      double res = sin(x/y) * 10;

      ch.SetDataY(row, SECOND_GRAPH, res);
    }

    ch.AutoScale();

    app.DelayMS(2000); // wait two seconds

    // Remove second graph
    ch.RemoveGraph(SECOND_GRAPH);

    ch.AutoScale();

    app.UpdateWindows();
  }
}
```

(from ..\Scriptings\Remove Graph.cpp)

Next example removes a Graph greater than a number.

```
#pragma extension "corelib"

#define STRING_NOT_FOUND -1

void main()
{
  Application app("My App");

  const string filenameEval = "H:/test.sx";

  if (app.FileExist(filenameEval))
  {
    app.LoadEval(filenameEval);

    Chart ch = app.GetChart("My Dummy Chart");

    for (int i = ch.GetNumberOfGraphs() - 1; i >= 0; i--)
    {
      if (app.EscapeLoop()) // Press key Esc and leave the loop
```

```
        break;

      string graphName = ch.GetGraphName(i);
      app.Output("Graph: " + graphName);

      string sub = graphName.substr(2, 4);

      int val = parseInt(sub);

      if (val > 2)
      {
        app.Output("Remove Graph: " + graphName);

        ch.RemoveGraph(i);
      }
    }

    app.UpdateWindows();
  }
  else
  {
    app.Error("File not found:" + filenameEval);
  }
}
```

(from ..\Scriptings\Remove Greater than.cpp)

2.8 Export Graphic as Image

Objectives:

1. Load any Evaluation
2. Get the chart object
3. Loop through a number of graphs
4. Export Chart Object as Bitmap

To export a graphic via script in form of selectable objects (see Layer selectable) use the next function:

Definition:

```
void ExportChartObjectAsBitmap(bool allObjects, string fileName);
```

Return Value:

nothing

Arguments:

`allObjects = true`	then the program selects all selectable objects, before it saves to disk
`allObjects = false`	the program saves all (already) selected objects to disk
`fileName`	path and filename for the image

Here the example script to store selected graphics objects as an image to disk.

```
#pragma extension "corelib"

#define NUMBER_OF_GRAPHS 10

void main()
{
  Application app("Simple App");

  string evalFilename = "e:/test/Surface Plot.sx";

  string exportFilename = "E:/test/Images/Image123.png";

  if (app.FileExist(evalFilename))
  {
    app.LoadEval(evalFilename);

    Chart ch = app.GetChart("MyChart");

    for (int graph = 1; graph < NUMBER_OF_GRAPHS; graph++)
    {
      if (app.EscapeLoop()) // Press key Esc and leave the loop
        break;

      if (!app.IsGraphicsViewAvailable()) // Do NOT forget this!
        break;
```

```
      string ext = formatInt(graph);

      if (graph < = 9)
        ext = "0" + ext;

      string newFilename = exportFilename.replace("123", ext, true);

      ch.SetActiveGraph(graph);

      app.DelayMS(300);

      app.ExportChartObjectAsBitmap(true, newFilename);
    }
  }
  else
  {
    app.Error("File does not exist!");
  }
}
```

(from ..\Scriptings\Export Chart as Bitmap.cpp)

2.9 Set Label

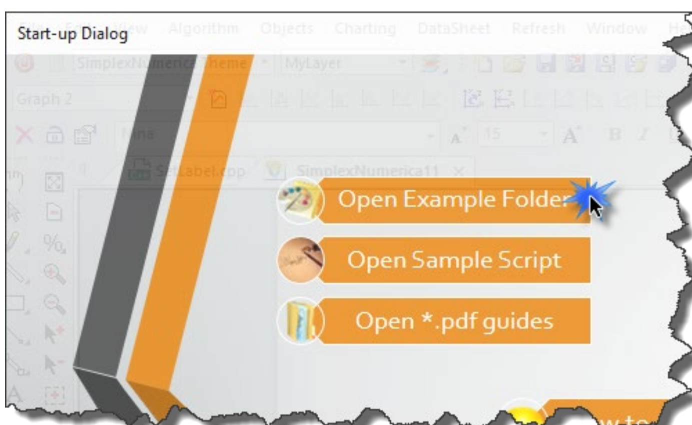
Objectives:

1. Load an Evaluation
2. Write a Text Label
3. Fit its Context
4. Change Text Color

In this example, we want to change the text of an existing ***Text Label***. To demonstrate this, a demo evaluation will be loaded, first.

Here the whole script code for doing this:

```
#pragma extension "corelib"

void main()
{
  Application app("My App");

  string simplexAppPath = app.GetSimplexAppPath();
  string filename = simplexAppPath + "Examples\\Meter\\Chart And Meter.sx";

  if (app.FileExist(filename))
  {
    app.LoadEval(filename);

    string label = "Label underneath Round Meter";
    string text = "This is my Round Meter!";

    // Write Label Text
    TShape sh = app.WriteTextLabel(label, text, true);

    app.FitContent(sh, 3);

    // Set Text Color
    app.SetColor(ID_PROP_TEXTCOLOR, 200, 0, 0, 255);
  }
}
```



All evaluation examples will be found in the setup folder `<..\Examples>`. You can easily have access to this folder by using the start-up dialog (press key <F1>).

```
string simplexAppPath = app.GetSimplexAppPath();
```

To find the example folder via script code, we need a function to tell us the root setup folder of ***SimplexNumerica***.

Definition:

```
string GetSimplexAppPath()
```

Return Value:

File path of the ***SimplexNumerica's*** setup folder.

Example:

```
string filename = simplexAppPath + "Examples\\Meter\\Chart And Meter.sx";
```

Concatenate setup-path and evaluation filename to the whole path and filename (double backslash!).

```
if (app.FileExist(filename))
```

It is always a good practice to proof on file exists before you load an (possibly not existing) evaluation file.

Definition:

```
bool FileExist()
```

Return Value:

true if file exist, else ***false***.

```
app.LoadEval(filename);
```

This is the application member function to load an evaluation from disk. There are to spellings available:

Definition:

```
void LoadEval(string filename)
```

or

```
void LoadEvaluation(string filename)
```

Argument:

Variable	Function
filename	Path and file name of the evaluation.

Example:

```
app.LoadEvaluation(simplexAppPath + "Examples\\Curve Fit\\Gauss-Fit.sx");
```

→ The backslash is doubly to set.

```
Important:
A backslash in explicit strings has to be put twice behind, like
c:\\MyPath or you can use merely a slash, instead: c:/MyPath
```

```
TShape sh = app.WriteTextLabel(label, text, true);
```

Use this member function of the application class to set another ***text*** in an existing ***Text Label***.

Definition:

```
TShape WriteTextLabel(string LabelName, string LabelText, bool select)
```

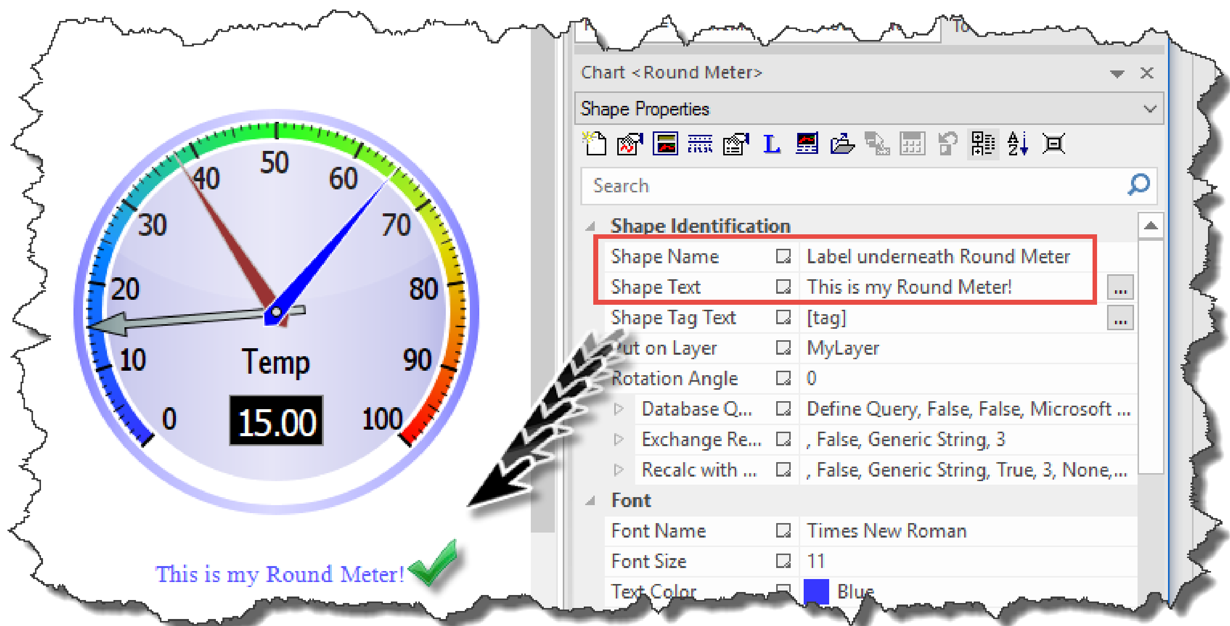
Return Value:

An instance of the Text Label object. If not found, then an error goes into the ***Output Window***.

Argument:

Variable	Function
LabelName	The Text Label Shape Name.
LabelText	The Text Label Text itself.
select	Select the text label afterwards (true or false).

Example:




```
// Write Label Text

sh = app.WriteTextLabel("Label underneath Round Meter", "This is my Round Meter!", false);
```

If the new text is longer than the old one, then this has to be adjusted to its new length. The next function does this.

```
app.FitContent(sh, 3);
```

This member function of the application class fits the content of a ***Text Label***. The last argument can be used to find the tightest bound of the text.

Definition:

```
void FitContent(TShape shape, uint wrapper)
```

Argument:

Variable	Function
shape	Shape object. Got from: `TShape sh = app.WriteTextLabel(..)`
`wrapper`	wrapper = 0: Text will be drawn automatically formatted. wrapper = 1: The height will not be increased. wrapper = 2: The width will not be increased. wrapper = 3: Increase width and height.

Example:

```
//..
string text = "This is a sentence no. 1 to format.\n";
text += "This is a sentence no. 2 to format.\n";
text += "This is a very long sentence no. 2223343434343 to format.";

// Write Label Text
TShape sh = app.WriteTextLabel(label, text, true);

app.FitContent(sh, wrapper);

//..
```

wrapper = 0

This is a sentence no. 1 to format.
This is a sentence no. 2 to format.
This is a very long sentence no. 2223343434343 to format.

wrapper = 1

This is a sentence no. 2 to format.

wrapper = 2

This is a sentence no. 1 to format. This is a sentence no. 2 to format. This is a very long sentence no. 2223343434343 to format.

wrapper = 3

This is a sentence no. 1 to format.
This is a sentence no. 2 to format.
This is a very long sentence no. 2223343434343 to format.

```
app.SetColor(ID_PROP_TEXTCOLOR, 200, 0, 0, 255);
```

Set the ***Text Color*** of all the selected ***Text Labels*** that have the property identifier ***ID_PROP_TEXTCOLOR***.

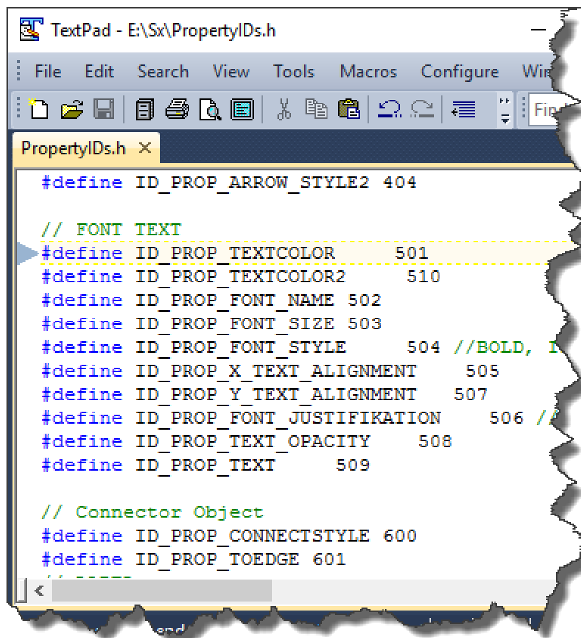
Definition:

```
void SetColor(uint Id, uint opacity, uint R, uint G, uint B)
```

Argument:

Variable	Function
Id*	Property Identifier
opacity	Opacity of the color (from 0 .. 255)
R	Red part of the color (from 0 .. 255)
G	Green part of the color (from 0 .. 255)
B	Blue part of the color (from 0 .. 255)

*As already explained in the first chapter, the identifiers were loaded with the header file



`<PropertyIDs.h>` that you can find in the root setup folder of ***SimplexNumerica***.

It defines each available property identifier (Id) of the ***Property Windows*** list, also ***ID_PROP_TEXTCOLOR***.

These Ids can be used in corresponding script functions, like this one here:

```
app.SetColor(ID_PROP_TEXTCOLOR,..);
```

That's it!

You can find this script file in your ***SimplexNumerica*** setup folder: `<..\Scriptings\Set Label.cpp>`

2.10 Arrange Charts

Objectives:

1. `#define` something simple
2. Select a chart
3. Move a chart
4. Copy & Paste a chart
5. Arrange two charts (or more)

To arrange charts on the evaluation page, we need at least two of them. Here, we will load an evaluation that contents only one chart. Then we will clone it by copy and paste. After that they are getting arranged.

Here the whole script code for doing the objectives:

```
#pragma extension "corelib"

#define TEST "Test"

void main()
{
  Application app("Simple App");

  string path = app.GetSimplexAppPath();

  string filename = path + "Examples\\Curve Fit\\Cyclic Smoothing Spline.sx";
  app.Output(filename);

  string chartname = TEST;

  app.LoadEval(filename);
  app.SelectChart(chartname);
  app.MoveChart(30, 30);
  app.SizeChart(400, 300);
  app.CopyChart();
  app.PasteChart();
  app.ArrangeCharts(10);
}
```

```
#define TEST "Test"
```

This is a common preprocessor definition, how it is often used in the C/C++ language. Other preprocessor definitions are:

```
#include

#define

#ifdef
 //..
#endif
```

```
#ifndef
 //..
#endif

#pragma

#warning
```

Example:

```
string chartname = TEST;
```

```
app.SelectChart(chartname);
```

Use this function to (visible) select a chart. Tell the program which chart you mean. If the program has already selected the chart, then it does nothing.

Definition:

```
void SelectChart(string name)
```

Arguments:

Variable	Function
Name	Name of the chart, that you want to select.

```
app.MoveChart(30, 30);
```

Move the selected chart to another x/y position.

Definition:

```
void MoveChart(int xPos, int yPos)
```

Arguments:

Variable	Function
xPos	x-Position of the top left chart corner.
yPos	y-Position of the top left chart corner.

```
app.SizeChart(400, 300);
```

Resize the selected chart to another outer width and height.

Definition:

```
void SizeChart(int width, int height)
```

Arguments:

Variable	Function
`width`	New width of the chart.
`height`	New height of the chart.

```
app.CopyChart();
```

Make a copy of the selected chart(s) to the windows clipboard.

```
void CopyChart()
```

```
app.PasteChart();
```

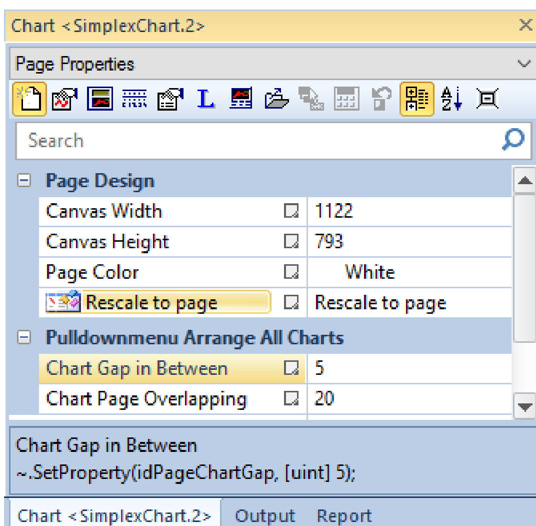
Paste the copied chart(s) from the windows clipboard to the evaluation page.

```
void PasteChart()
```

```
app.ArrangeCharts(10);
```

In conjunction with the Pulldownmenu ***Arrange All Charts*** to arrange all charts (toolbar icon) you can use this function from the scripting host.

Before you use this function, try it manually to rearrange, because if the page is too small, then the program stops with the arrangement and writes an error in the ***Output Window***. Go to the ***Property Window*** and select ***Page Properties***. Then set the two parameters to your own values:



→ Gap between the charts.
→ Chart page overlapping.

Definition:

```
void ArrangeCharts(int gap)
```

Arguments:

Variable	Function
gap	Width of the gap between two charts.

That's it!

You can find this script file in your ***SimplexNumerica*** setup folder:

<..\Scriptings\ ArrangeCharts.cpp>

2.11 Set Property

Objectives:

1. Load an Evaluation
2. Save an Evaluation
3. Set Properties
4. Close an Evaluation

In this example, we want to repeat some in the last chapters discussed functionalities. The focus here again lies on evaluations and properties.

Here the whole script code for doing the objectives:

```
#pragma extension "corelib"

#define IDYES               6
#define IDNO                7

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetSimplexAppPath();

  string filename = simplexAppPath + "Examples\\Curve Fit\\Gauss-Fit.sx";
  app.Output(filename);

  if (app.FileExist(filename))
  {
    app.LoadEval(filename);

    Chart ch = app.MakeChart("My Chart", idChartTypePhysics, 90, 70, 400, 300);
    ch.SelectPropertyGroup("Chart Properties");
    ch.SetProperty(idShowGrid, false);

    app.SelectPropertyGroup("Page Properties");
    app.SetProperty(idShowPageGrid, true);
    app.SelectChart("My Chart");
    app.SaveEval(simplexAppPath + "test.sx");

    if (alertYes("Close Evaluation?") == IDYES)
    {
      app.CloseEval();
    }
  }
  else
  {
    app.Error("File does not exist!");
  }
}
```

```
app.LoadEval(filename);
```

This is the application member function to load an evaluation from disk. There are to spellings available:

Definition:

```
void LoadEval(string filename)
```

or

```
void LoadEvaluation(string filename)
```

Argument:

Variable	Function
filename	Path and file name of the evaluation.

```
app.SaveEval(filename);
```

This is the application member function to save an evaluation to disk. There are to spellings available:

Definition:

```
void SaveEval(string filename)
```

or

```
void SaveEvaluation(string filename)
```

Argument:

Variable	Function
filename	Path and file name of the evaluation.

```
app.CloseEval(filename);
```

This is the application member function to close an evaluation. There are to spellings available:

Definition:

```
void CloseEval()
```

or

```
void CloseEvaluation()
```

Example from the code further up:

```
if (alertYes("Do you want to close your evaluation?") == IDYES)
{
   app.CloseEval();
}
```

We have to ask the user via script code whether to close or not the page.

```
app.SelectPropertyGroup("Page Properties");
```

This member function of the ***Application*** class selects the ***Property Window Group*** by name. If the property group is not related to a chart, then you should use this instead of the same Chart member function.

Please use for charts

```
ch.SelectPropertyGroup("Chart Properties");
```

else use e.g.

```
app.SelectPropertyGroup("Page Properties");
```

Definition:

```
void SelectPropertyGroup(string group)
```

Arguments:

Variable	Function
group	Property Group Name

```
app.SetProperty(idShowPageGrid, true);
```

This member function of the ***Application*** class selects the property itself. Again, if the property group is not related to a chart, then you should use this instead of the same Chart member function.

Please use for charts

```
ch.SetProperty(idShowGrid, false);
```

else use e.g.

```
app.SetProperty(idShowPageGrid, true);
```

Definition:

```
void SetProperty(uint Id, bool var)
```

or overloaded:

```
void SetProperty(uint Id, string var)
```

Arguments:

Variable	Function
Id	Identifier from the file <PropertyIDs.h>
var	Value to set into the property cell.

That's it!

You can find this script file in your ***SimplexNumerica*** setup folder:

<..\Scriptings\ SetProperty.cpp>

2.12 Load Project

This is a small example; it only loads a project. You can find this script file in your ***SimplexNumerica*** setup folder: `<..\Scriptings\Load Project.cpp>`.

Here the whole script code for doing this:

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetSimplexAppPath();

  string filename = simplexAppPath + "Projects\\Smmothing Splines.sxw";
  app.Output(filename);

  if (app.FileExist(filename))
  {
    app.LoadProject(filename);
  }
  else
  {
    app.Error("File does not exist!");
  }
}
```

Definition:

```
void SaveEval(string filename)
```

This is the application member function to load a project from disk with a ***string*** argument.

Variable	Function
filename	Path and file name of the project.

2.13 Import and Calc Data

This is an important example to learn data import via scripting host.

Objectives:

1. Load an existing evaluation
2. Select its only chart
3. Duplicate the chart with Copy & Paste
4. Arrange and AutoScale the two charts
5. Set a y-Axis to logarithmic scale
6. Set the CSV settings in a separate C++ function
7. Load the CSV file
8. Manipulate the graph data and write it back to the chart memory

→ For the first objectives, please have a look to the previous chapters…

Here the whole script code for doing this:

```
#pragma extension "corelib"

double __min(double a, double b)
{
  return ((a < b) ? a : b);
}

double __max(double a, double b)
{
  return ((a > b) ? a : b);
}

void main()
{
  Application app("My App");

  string simplexAppPath = app.GetSimplexAppPath();

  string filename = simplexAppPath + "Examples\\DataPlots\\Spectrum Data.sx";
  app.Output(filename);

  if (app.FileExist(filename))
  {
    // Load an evaluation
    app.LoadEval(filename); // A chart with the name "First Chart"

    // Make a second chart similar to "First Chart" and call it "Second Chart"
    app.SelectChart("First Chart");
    app.CopyChart();
    app.PasteChart();
    app.ArrangeCharts(10);

    // Copy & Paste a chart brings up an index behind the copied name
    Chart ch2 = app.GetChartByName("First Chart.1");
```

```
    // Rename second chart
    ch2.SetName("Second Chart");

    // Get the first chart object
    Chart ch = app.GetChartByName("First Chart");

    // Set the CSV Import Dialog parameter (see function below on this page)
    SetCSVSettings(ch);

    // Import any data from a CSV file!
    ch.LoadCSV(simplexAppPath + "Data\\Sample3.csv");

    // Manipulate the data and write it back to the chart memory
    for (int i = 0; i < ch.GetNumberOfSampleData(0); i++)
    {
      int graph = 0; // first graph
      double y = ch.GetDataY(i, graph);

      y *= 100 / sqrt(2); // Calc anything

      ch.SetDataY(i, graph, y);
    }

    // Now, write the y data from second chart to a script array
    array<float> ay(ch2.GetNumberOfSampleData(0)); // 0 = Graph No. 0

    for (int j = 0; j < ch2.GetNumberOfSampleData(0); j++)
    {
      ay[j] = ch2.GetDataY(j, 0);
    }

    // Next, add this data to the first charts graph data

    int Nd = __min(ch.GetNumberOfSampleData(0), ch2.GetNumberOfSampleData(0));

    for (int i = 0; i < Nd; i++)
    {
      double y = ch.GetDataY(i, 0);

      y += ay[i];

      ch.SetDataY(i, 0, y);
    }

    ch.SetLogScaleY(true);
    ch.AutoScale();

    // Finally update properties on screen
    app.UpdateWindows();
  }
  else
  {
    app.Error("Could not find the chart");
  }
}
```

```
void SetCSVSettings(Chart& ch)
{
  // ==========================================
  // CSV Parameter, Dump for Scripting Host
  // Made by button <Script Dump to Clipboard>
  // at the bottom of the Import Dialogbox
  // ==========================================

  ch.SetColumnsSeparation(4);
  ch.SetDecimalSeparation(2);
  ch.SetOrderAxesToColumns(1);
  ch.SetAppendToGraphMemory(false);
  ch.SetJumpOverFirstNRows(false);
  ch.SetJumpOverFirstRow(false);
  ch.SetJumpOverSecondRow(false);
  ch.SetPutFirstColumnInLegend(false);
  ch.SetbSetNextColumnForAllOtherXAxis(false);
  ch.SetExpectingMissingValues(false);
  ch.SetGraphNameFromFirstRow(false);
  ch.SetHeaderNameFromFirstRow(false);
  ch.SetSkipOverEachMRow(false);
  ch.SetUseAveraging(false);
  ch.SetJumpOverNumberOfStartRows(3);
  ch.SetSkipOverNumberOfRows(2);
  ch.SetJumpOverNumberOfHeaderRows(0);
  ch.SetJumpOverFirstNHeaderRows(false);
}
```

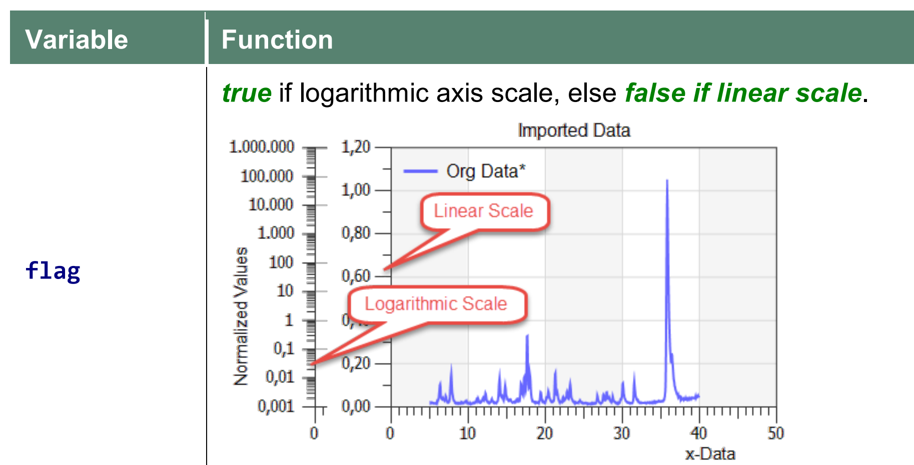
```
ch.SetLogScaleY(true);
```

Sets the ordinate axis to a logarithmic or linear scale. Default is linear scale for all charts in ***SimplexNumerica***.

Definition for y-Axis:

```
void SetLogScaleY(bool flag)
```

Argument:

<table>
<tr><th>Variable</th><th>Function</th></tr>
<tr><td>flag</td><td>true if logarithmic axis scale, else false if linear scale.

</td></tr>
</table>

```
ch.SetLogScaleX(true);
```

Sets the abscissa axis to a logarithmic or linear scale. Default is linear scale for all charts in ***SimplexNumerica***.

Definition for x-Axis:

```
void SetLogScaleX(bool flag)
```

Argument:

Variable	Function
flag	***true*** if logarithmic axis scale, else ***false if linear scale***.

Definition:

```
void AutoScale()
```

AutoScale all chart axes.

```
ch.SetCSVSettings(ch);
```

This is the call for the separate function at the bottom of this example script file.

```
void SetCSVSettings(Chart& ch)
{
  // =========================================
  // CSV Parameter, Dump for Scripting Host
  // Made by button <Script Dump to Clipboard>
  // at the bottom of the Import Dialogbox
  // =========================================

  ch.SetColumnsSeparation(4);
  ch.SetDecimalSeparation(2);
  ch.SetOrderAxesToColumns(1);
  ch.SetAppendToGraphMemory(false);
  ch.SetJumpOverFirstNRows(false);
  ch.SetJumpOverFirstRow(false);
  ch.SetJumpOverSecondRow(false);
  ch.SetPutFirstColumnInLegend(false);
  ch.SetbSetNextColumnForAllOtherXAxis(false);
  ch.SetExpectingMissingValues(false);
  ch.SetGraphNameFromFirstRow(false);
  ch.SetHeaderNameFromFirstRow(false);
  ch.SetSkipOverEachMRow(false);
  ch.SetUseAveraging(false);
  ch.SetJumpOverNumberOfStartRows(3);
  ch.SetSkipOverNumberOfRows(2);
  ch.SetJumpOverNumberOfHeaderRows(0);
  ch.SetJumpOverFirstNHeaderRows(false);
}
```

This is the function itself. It defines the settings for the data import

Hint
These settings came from the data import dialog via Copy & Paste.
→ Please have a look to the chapter "Import CSV File" in the main manual.

```
app.LoadCSV(filename);
```

This is the application member function to load a **C**omma **S**eparated **F**ile (CSV) with any extension (*.csv is certainly preferred) from disk.

Definition:

```
void LoadCSV(string filename)
```

Argument:

Variable	Function
filename	Path and file name of the CSV file.

2.13.1 Manipulate sample data and write it back to the chart memory

We like to descripe this part of the code in blocks so that we can understand the context.

Objectives Block 1:

1. Get ***SampleData*** from first ***Graph***
2. Manipulate the data
3. Set manipulated data back into ***SampleData*** from first ***Graph***

...certainly, you can use arrays to store data, but here we want to use simply a double value.

```
for (int i = 0; i < ch.GetNumberOfSampleData(0); i++)
{
  int graph = 0; // first graph
  double y = ch.GetDataY(i, graph);

  y *= 100 / sqrt(2); // Calc anything

  ch.SetDataY(i, graph, y);
}
```

Definition:

```
int GetNumberOfSampleData(int graph)
```

Return Value:

Number of SampleData inside this graph

Argument:

Variable	Function
graph	Graph number (i = 0, 1, .. n – 1)

Definition:

```
int GetNumberOfGraphs()
```

Return Value:

Number of Graphs inside this chart

Definition:

```
void SetNumberOfSampleData(int N, int graph)
```

Argument:

Variable	Function
N	Number of SampleData to set for this graph
graph	Graph number (g = 0, 1, .. m – 1)

Definition:

```
double GetDataX(int index, int graph)
double GetDataY(int index, int graph)
double GetDataZ(int index, int graph)
```

Return Value:

SampleData real value on index position for this graph

Argument:

Variable	Function
index	SampleData point (i = 0, 1, .. n – 1)
graph	Graph number (g = 0, 1, .. m – 1)

Definition:

```
void SetDataX(int index, int graph, double value)
void SetDataY(int index, int graph, double value)
void SetDataZ(int index, int graph, double value)
```

Argument:

Variable	Function
index	SampleData point (i = 0, 1, .. n – 1)
Graph	Graph number (g = 0, 1, .. m – 1)
value	*SampleData* real value on index position for this graph

Objectives Block 2:

1. Write the y data from second chart to an internal script array
2. Next, add this data to the first charts graph data
3. Finally, update properties on screen

Here the second code block from the "Import and Calc Data" script file:

```
// Now, write the y data from second chart to a script array
array<float> ay(ch2.GetNumberOfSampleData(0)); // 0 = Graph No. 0

for (int j = 0; j < ch2.GetNumberOfSampleData(0); j++)
{
  ay[j] = ch2.GetDataY(j, 0);
}

// Next, add this data to the first charts graph data

int Nd = __min(ch.GetNumberOfSampleData(0), ch2.GetNumberOfSampleData(0));

for (int i = 0; i < Nd; i++)
{
  double y = ch.GetDataY(i, 0);

  y += ay[i];

  ch.SetDataY(i, 0, y);
}
```

`array<float> ay(n)`

When declaring dynamic arrays in *AngelScript,* it is possible to define the initial size of the array by passing the length as a parameter to the constructor.

Reference

Please have a look to the chapter "AngelScript" → "Template Arrays" in this manual at chapter 6.4.

That's it!

You can find this script file in the setup folder: `<..\Scriptings\ Import and Calc Data.cpp>`

2.14 Make Text Label

Objectives:

1. Create a new shape, a ***Text Label***
2. Change its style
3. Fit its Context
4. Move it around

A Text Label is a special text shape in *SimplexNumerica* that has much more power as it looks like (e.g. text rendering, database support, report functionality, etc.)

→ Here, we do not use the ***SetProperty(..)*** function from above, we use instead the member functions from the ***TShape*** class.

→ This a sequel to the chapter *2.9 Set Label*. In contrast to this chapter, where existing ***Text Shapes*** were manipulated, we will make a new one here and move it horizontal from left to right.

The whole script code make a Text Label and to change its style:

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  app.NewEval();

  string shapeName = "My Princesses";
  string shapeText = "Denise und Estelle";

  float x1 = 100.0;
  float y1 = 70.0;
  float width = 0.0; // set to 0 means auto-fit size
  float height = 0.0;

  TShape sh = app.MakeTextLabel(shapeName, shapeText, x1, y1, width, height);

  // Switch off the redraw for this shape for its next set functions (faster)
  //sh.RedrawOff();

  // ...or use this function for deactivating redraw
  app.RedrawOff(sh);

  // Call again - shape will be selected, only!
  // sh = app.MakeTextLabel(shapeName, shapeText, x1, y1, width, height);

  // Change Text Color (Not necessary too select the shape, before!)
  // SetTextColor(int opacity, int R, int G, int B);
  // SetTextColor(int R, int G, int B);
  // SetTextColor(int RGB);
  sh.SetTextColor(255, 0, 0);
```

```
  // Change the font name
  sh.SetFontName("Times New Roman");

  // Change the font size
  sh.SetFontSize(36);

  // Set Font Style
  bool bBold = true;
  bool bItalic = false;
  bool bUnderline = true;
  bool bStrikethrough = false;
  sh.SetFontStyle(bBold, bItalic, bUnderline, bStrikethrough);

  // Set Font Alignment (x, y)
  // x Direction:
  //   Left = 0
  //   Center = 1
  //   Right = 2
  // y Direction:
  //   Top = 0
  //   Center = 1
  //   Bottom = 2
  sh.SetFontAlignment(1, 0);

  // Set Font Justification (0: Right-to-left or 1: Vertical)
  sh.SetFontJustification(0);

  // Set Font Opacity ( 0 - 255)
  sh.SetFontOpacity(200);

  // Already done above in <MakeTextLabel(..)>
  // sh.SetText(shapeText);

  // Find the tightest bound of text
  int orientation;
  // orientation = 0; // Text will be drawn without wrapping
  // orientation = 1; // Height will not be increased
  // orientation = 2; // Width will not be increased
  orientation = 3; // increase width and height
  app.FitContent(sh, orientation);

  // Switch on the redraw and program makes a redraw
  // sh.RedrawOn(); // Redraw only the shape <sh>!

  // ...or use this function for activating redraw
  app.RedrawOn(sh); // Redraw the whole graphics view and not only the shape
<sh>!

  app.Output("Begin");

  // Move the text on the graphics screen, go out when the window was closed
  for (int i = 0; i < 100; i += 1)
  {
    app.DelayMS(20);
```

```
    // Do NOT forget this in a loop that outputs graphic!
    if (!app.IsGraphicsViewAvailable())
      break;

    sh.MoveTo(i / 2, 150);
  }

  app.Output("End");
}
```

```
TShape sh = app.MakeTextLabel(shapeName, shapeText, x1, y1, width, height);
```

Use this member function of the application class to make a new ***Text Label***.

Definition:

```
TShape MakeTextLabel(string label, string text, float x, float y, float w, float h)
```

Return Value:

An instance of the Text Label object. If not been made, then an error goes into the ***Output Window***.

Argument:

Variable	Function
label	The Text Label Shape Name.
text	The Text Label Text itself.
x	x-Position of the top left corner
y	y-Position of the top left corner
w	Width of the outer shape
h	Height of the outer shape

```
Hint
When you call this function again, with the same label name, then it will
be selected and nothing else.
```

```
// Switch off the redraw for this shape for its next set functions (faster)
sh.RedrawOff();

// ...or use this function for deactivating redraw
app.RedrawOff(sh);
```

There are two functions, with the same purpose available, to avoid each time a redraw of the object or whole page. That makes sense, when you want to change more than two properties in addition.

Definition for the ***Application*** class member function:

```
void RedrawOff()
void RedrawOn()
```

No return value and no argument.

Definition for the ***TShape*** class member function:

```
void RedrawOff(TShape sh)
void RedrawOn(TShape sh)
```

Argument:

Variable	Function
sh	The ***TShape*** object name, returned in MakeTextLabel(..)

To switch on the redraw please use the *RedrawOn(..)* functions. The program initiated immediately a redraw. A good place for this is at the end of the ***main*** function.

```
Hint
The following TShape member functions can be used without to select the
Text Shape.
```

2.14.1 Change Text Color

There are three overloaded TShape member functions available to change the text color of the *Text Shape*.

```
void SetTextColor(uint opacity, uint R, uint G, uint B)
void SetTextColor(uint R, uint G, uint B)
void SetTextColor(uint RGB)
```

Argument:

Variable	Function
Id*	Property Identifier
opacity	Opacity of the color (from 0 .. 255)
R	Red part of the color (from 0 .. 255)
G	Green part of the color (from 0 .. 255)
B	Blue part of the color (from 0 .. 255)

Variable	Function
RGB	(Hex) value of the color

2.14.2 Change Font Name

```
void SetFontName(string fontName)
```

Argument:

Variable	Function
fontName	That's the windows font name, like ***Arial*** or ***Times New Roman***

2.14.3 Change Font Size

```
void SetFontSize(uint size)
```

Argument:

Variable	Function
size	That's the font size as a number, e.g. 12 pixel

2.14.4 Change Font Style

```
void SetFontStyle(bool bBold, bool bItalic, bool bUnderline, bool bStrikethrough)
```

Argument:

Variable	Function
…	As you know it inside out.

2.14.5 Change Font Alignment

```
void SetFontAlignment(uint xAlign, uint yAlign)
```

Argument:

Variable	Function

xAlign	Font alignment in x direction
	// Left = 0
	// Center = 1
	// Right = 2

Variable	Function

yAlign	Font alignment in y direction
	// Top = 0
	// Center = 1
	// Bottom = 2

2.14.6 Change Font Justification

```
void SetFontJustification(uint just)
```

Argument:

Variable	Function
Just	Justification of the text. 0: Right-to-left or 1: Vertical

2.14.7 Change Font Opacity

```
void SetFontOpacity(uint opacity)
```

Argument:

Variable	Function
opacity	Opacity of the color (from 0 .. 255)

2.14.8 Change Text itself

```
void SetText(string shapeText)
```

Argument:

Variable	Function
shapeText	Text of the label. → Already done above in ***MakeTextLabel(shapeName, shapeText, ..)***

2.14.9 Move any Shape

You can move any shape via script on your Graphics screen. Important is, that the window still exists as long as you moving around with your shape.

→ If you manipulate your shape in a (endless) loop, then check whether your Graphics window is still there!

```
app.Output("Begin");

// Move the text on the graphics screen, go out when the window was closed
for (int i = 0; i < 100; i += 1)
{
  app.DelayMS(20);


  // Do NOT forget this in a loop that outputs graphic!
  if (!app.IsGraphicsViewAvailable())
    break;

  sh.MoveTo(i / 2, 150);
}

app.Output("End");
```

We have in this code block three unknown functions to explain.

```
app.DelayMS(20);
```

Use this member function of the application class to wait a certain time in milliseconds (ms) without to interrupt the whole program.

Definition:

```
void DelayMS(uint ms)
```

Argument:

Variable	Function
ms	Waiting time in milliseconds.

```
if (!app.IsGraphicsViewAvailable())
  break;
```

Use this member function of the application class to check whether the ***Graphics View*** is available. Meant is here your evaluation page.

Logical Fact

You cannot manipulate a shape on a not existing evaluation page.

Definition:

```
bool IsGraphicsViewAvailable()
```

Return Value:

true for is available else ***false***

```
sh.MoveTo(i / 2, 150);
```

That's ***TShape*** member function that moves any shape around the screen on the graphics page. Please use the previous function `IsGraphicsViewAvailable()` to check whether the window was not interim closed.

Definition:

```
void MoveTo(uint xPos, uint yPos)
```

Argument:

Variable	Function
xPos	Position in x direction on the Graphics page.
yPos	Position in y direction on the Graphics page.

That's it!

You can find this script file in the setup folder: <..\Scriptings\Make Text Label.cpp>

2.15 Make Drawing Shape

Objectives:

1. Create a new ***Drawing Shape***
2. Change its style

A ***Drawing Shape*** is functional similar to a ***Text Label***, or better to say, internally, a ***Text Label*** based on a ***Drawing Shape.*** Vice versa, a ***Drawing Shape*** can have text within**.**

→ Rectangles, circles, polygons, etc. are ***Drawing Shapes***!

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  app.NewEval();

  string shapeName = "My Ellipse";

  DShape ds = app.MakeDrawingShape(shapeName, "Ellipse", 100, 50, 80, 70);

  /*
    float w = ds.GetWidth();
    float h = ds.GetHeight();

    ds.SetWidth(2 * w);
    ds.SetHeight(2 * h);

    ds.MoveTo(0,0);
   */

  ds.SetLineColor(255, 255, 0, 0);
  ds.SetFillColor(255, 0, 250, 0);
}
```

```
  DShape ds = app.MakeDrawingShape(shapeName, "Ellipse", 100, 50, 80, 70);
```

Use this member function of the application class to make a new Drawing Shape ***(DShape)***.

Definition:

```
DShape MakeDrawingShape(string shapeName, string objectName,
                        float x, float y, float w, float h)
```

Return Value:

An instance of the DShape object. If not been made, then an error goes into the ***Output Window***.

Argument:

Variable	Function
label	The Text Label Shape Name.
text	The Text Label Text itself.
x	x-Position of the top left corner
y	y-Position of the top left corner
w	Width of the outer shape
h	Height of the outer shape

```
ds.SetLineColor(255, 255, 0, 0);
```

This is a ***DShape*** member function to change the ***line color*** of a ***Drawing Shape***.

```
void SetLineColor(uint opacity, uint R, uint G, uint B)
```

Argument:

Variable	Function
opacity	Opacity of the color (from 0 .. 255)
R	Red part of the color (from 0 .. 255)
G	Green part of the color (from 0 .. 255)
B	Blue part of the color (from 0 .. 255)

```
ds.SetLineColor(255, 255, 0, 0);
```

This is a ***DShape*** member function to change the ***fill color*** of a ***Drawing Shape***.

```
void SetFillColor(uint opacity, uint R, uint G, uint B)
```

Argument:

Variable	Function
opacity	Opacity of the color (from 0 .. 255)
R	Red part of the color (from 0 .. 255)
G	Green part of the color (from 0 .. 255)
B	Blue part of the color (from 0 .. 255)

That's it!

You can find this script file in the setup folder: `<..\Scriptings\Make Drawing Shape.cpp>`

2.16 Make Chart on Layer

Objectives:

1. Get the active layer.
2. Create a new chart.
3. Change name of the chart.
4. Get the active layer, again.

This example and the following chapters are concentrating on ***Layers*** in *SimplexNumerica*. Please have a look to the main manual to find out more about layers.

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  app.NewEval();

  // Ask for the layer name
  Layer layer = app.GetActiveLayer();

  string strActiveLayerName = layer.GetName();
  app.Output(alertOk("Name of the active layer: " + strActiveLayerName));

  Chart ch = app.MakeChart("My Chart", idChartTypePhysics, 100, 100, 400, 300);

  ch.SelectPropertyGroup("Chart Properties");
  ch.SetProperty(idChartName, "My Chart");

  ch.SetWidth(ch.GetWidth() * 2);

  app.SelectChart("My Chart");

  // Ask again for the layer name
  layer = app.GetActiveLayer();
  strActiveLayerName = layer.GetName();
  app.Output(alertOk("Name of the active layer\nafter MakeChart: " +
strActiveLayerName));

  // Finally update properties on screen
  app.UpdateWindows();
}
```

```
Layer layer = app.GetActiveLayer();
```

The main class for layers is so called ***Layer***. This member function of the ***Application*** class returns such a ***Layer*** instance, here the active layer. You can use this to call its member functions.

Definition:

```
Layer GetActiveLayer()
```

Return Value:

Returns a layer instance. An object variable, that can be used for the ***Layer*** member functions.

Remark:

The second call of the same function `GetActiveLayer()` demonstrates the fact that the chart function `MakeChart(..)` applies automatically a new layer on the evaluation page.

That's it!

You can find this script file in the setup folder: `<..\Scriptings\Make Chart on Layer.cpp>`

2.17 Update Layer Window

Objectives:

1. Update the Layer Window.
2. Update the Chart Explorer.
3. Update the Property Window.

Use this function at the end of the main function to update the relevant Objectives windows.

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  app.NewEval();

  // Ask for the layer name
  Layer layer = app.GetActiveLayer();

  string strActiveLayerName = layer.GetName();
  app.Output("Name of the active layer: " + strActiveLayerName);

  app.UpdateWindows();
}
```

```
app.UpdateWindows();
```

Use this short member function of the ***Application*** class to update the above descripted windows.

Definition:

```
void UpdateWindows()
```

2.18 Make Chart on Layer Extended

New Objectives:

1. Add a new Chart Layer.
2. Add any new Layer.
3. Use Layer member functions.
4. Activate a layer.

This is an extended chapter to the chapter 2.16.

```
#pragma extension "corelib"

#define IDOK                1
#define IDCANCEL            2
#define IDABORT             3
#define IDRETRY             4
#define IDIGNORE            5
#define IDYES               6
#define IDNO                7

void main()
{
  Application app("Simple App");

  app.NewEval();

  // Ask for the layer name
  Layer layer = app.GetActiveLayer();

  string strActiveLayerName = layer.GetName();
  app.Output("Name of the active layer: " + strActiveLayerName);

  /* Chart Type Ids
  idChartTypeBar
  idChartTypeContourPlot
  idChartTypeMath
  idChartTypeMeter
  idChartTypeMisc
  idChartTypePhysics
  idChartTypePie
  idChartTypePolarV2
  idChartTypeSmith
  idChartTypeSurface
  idChartTypeTriangle
  idChartTypeXYLine
  */

  string strMyChart = "My Chart";

  Chart ch = app.MakeChart(strMyChart, idChartTypePhysics, 100, 100, 400, 300);
```

```
ch.SelectPropertyGroup("Chart Properties");
ch.SetProperty(idChartName, "My Chart");
ch.SetWidth(ch.GetWidth() * 2);

app.SelectChart("My Chart");

// Ask again for the layer name
layer = app.GetActiveLayer();
strActiveLayerName = layer.GetName();
app.Output("Name of the active layer\nafter MakeChart: " + strActiveLayerName);

// Add a chart type layer
app.AddChartLayer(idChartTypeMeter);

string strMyNewLayer = "This is my new layer!";

// Add two named layer
app.AddLayer(strMyNewLayer);
app.AddLayer("Dummy Layer"); // This is now active!

// Activate my new layer
app.ActivateLayer(strMyNewLayer);

string strQuestion = "Would you like to remove the active layer?";

/*
  if (alertYes(strQuestion) == IDYES)
    app.RemoveActiveLayer();  // Remove "This is my new layer!"

  if (alertYes("Would you like to remove the dummy layer?") == IDYES)
    app.RemoveLayer("Dummy Layer"); // Remove this also
*/

// Activate my new layer again (If it was removed, then it is empty now!)
app.ActivateLayer(strMyNewLayer);

// Place "My Chart" to my new layer
app.PutObjectOnLayer(strMyChart, strMyNewLayer);

// Test the layer properties
layer = app.GetActiveLayer();

bool vis = layer.IsVisible();
vis ? app.Output("Layer is visible!") : app.Output("Layer is not visible!");

bool sel = layer.IsSelectable();
sel ? app.Output("Layer is selectable!") : app.Output("Layer is not selectable!");

bool inh = layer.IsInhibit();
inh ? app.Output("Layer is inhibit!") : app.Output("Layer is not inhibit!");

// Change the properties
layer.SetVisible(true);
layer.SetSelectable(false);
layer.SetInhibit(false);
```

```
  app.Output("Properties have changed!");

  vis = layer.IsVisible();
  vis ? app.Output("Layer is visible!") : app.Output("Layer is not visible!");

  sel = layer.IsSelectable();
  sel ? app.Output("Layer is selectable!") : app.Output("Layer is not selectable!");

  inh = layer.IsInhibit();
  inh ? app.Output("Layer is inhibit!") : app.Output("Layer is not inhibit!");

  // Finally update properties on screen
  app.UpdateWindows();
}
```

```
layer.GetName();
```

Call this member function of the ***Layer*** class to get back the name of the layer.

Definition:

```
string GetName()
```

Return Value:

Returns the name of the layer.

```
app.AddChartLayer(idChartTypeMeter);
```

This is the main member function of the ***Application*** class to add a new chart layer to the internal list of layers.

Definition:

```
void AddChartLayer(uint chartType)
```

Argument:

Variable	Function
chartType	That's one of the following chart types: idChartTypeBar idChartTypeContourPlot idChartTypeMath idChartTypeMeter idChartTypeMisc idChartTypePhysics idChartTypePie idChartTypePolarV2 idChartTypeSmith idChartTypeSurface idChartTypeTriangle idChartTypeXYLine

```
app.AddLayer("Dummy Layer"); // This layer is now active!
```

This is the main member function of the ***Application*** class to add any new layer to the internal list of layers.

Definition:

```
void AddLayer(string layerName)
```

Argument:

Variable	Function
layerName	Your preferred name for the new layer.

```
app. RemoveActiveLayer();
```

Call this member function of the ***Application*** class to remove the active layer.

Definition:

```
void RemoveActiveLayer()
```

```
app. RemoveLayer("Dummy Layer");
```

Call this member function of the ***Application*** class to remove the layer that has the name from the argument.

Definition:

```
void RemoveLayer(string layerName)
```

Argument:

Variable	Function
layerName	The layer name for the layer that should be removed.

```
app. PutObjectOnLayer(strMyChart, strMyNewLayer);
```

Call this member function of the ***Application*** class to put an object (shape/chart) on the specified layer.

Definition:

```
void PutObjectOnLayer(string objectName, string layerName)
```

Argument:

Variable	Function
objectName	The name of the object.

Variable	Function
`layerName`	The name of the layer.

```
layer.IsVisible();
```

Call this member function of the ***Layer*** class to inform about the layer is visible or not.

Definition:

```
bool IsVisible()
```

Return Value:

true if the layer is visible else ***false***.

```
layer.IsSelectable();
```

Call this member function of the ***Layer*** class to inform about the layer is selectable or not.

Definition:

```
bool IsSelectable()
```

Return Value:

true if the layer is selectable else ***false***.

```
layer.IsInhibit();
```

Call this member function of the ***Layer*** class to inform about the layer is inhibitit or not.

Definition:

```
bool IsInhibit()
```

Return Value:

true if the layer is inhibitit else ***false***.

```
layer.SetVisible();
```

Call this member function of the ***Layer*** class to set the layer visible or not.

Definition:

```
void SetVisible(bool flag)
```

Argument:

Variable	Function
flag	***true*** if the layer should be visible else ***false***

```
layer.SetVisible();
```

Call this member function of the ***Layer*** class to set the layer selectable or not.

Definition:

```
void SetSelectable(bool flag)
```

Argument:

Variable	Function
flag	***true*** if the layer should be selectable else ***false***

```
layer.SetVisible();
```

Call this member function of the ***Layer*** class to set the layer inhibit or not.

Definition:

```
void SetInhibit(bool flag)
```

Argument:

Variable	Function
flag	***true*** if the layer should be inhibit else ***false***

That's it!

You can find this script file in the setup folder:

`<..\Scriptings\Make Chart on Layer Extended.cpp>`

```
Hint
Please have a look at the example "Import and Calc on Layers.cpp".
For instance, you could import data from a CSV file on a chart that
overlies a hidden layer. Then manipulate the data on that chart and copy
the results to another chart mainly on your focus.
```

2.19 Write to Excel File

Objectives:

1. Establish the Microsoft Excel Interface.
2. Load Excel File.
3. Add Excel Sheet.
4. Write a String to the Excel file
5. Write a Number to the Excel file
6. Save Excel File.
7. Release the Interface

These functions from the excel interface in *SimplexNumerica* are very straightforward to use. Maybe you can already imagine what each one means…

Hint

Be sure that Excel does not have open your *.xls(x) file! → Only one application can manipulate the file, concurrently!

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetSimplexAppPath();
  string filename = simplexAppPath + "Data\\CubicSpline Grid Points.xlsx";
  app.Output(filename);

  // Microsoft Excel Interface
  ExcelInterface excel("xlsx"); // File Extension *.xls or *.xlsx

  excel.LoadExcelFile(filename);

  excel.AddExcelSheet("My Sheet"); // Name of the sheet

  excel.WriteStringToExcel(2, 0, "Hello Excel!"); // row, column, text
  excel.WriteNumberToExcel(3, 1, 3.14); // row, column, double value

  // File Extension must be the same as before!
  excel.SaveExcelFile(simplexAppPath + "Data\\My Test Excel File.xlsx");

  excel.ReleaseInterface(); // Important to do!
}
```

```
ExcelInterface excel("xlsx"); // File Extension *.xls or *.xlsx
```

This is the declaration of the ***Excel interface*** in *SimplexNumerica.* The lower case word “`excel`” is your free instance name. The extension should be either `"xls"` or `"xlsx"`, for old or new format.

```
excel.LoadExcelFile(filename);
```

Use this member function of the **ExcelInterface** class to load an Excel file.

Definition:

```
void LoadExcelFile(string fileName)
```

Argument:

Variable	Function
fileName	The full path plus filename of the Excel file. Please use only Excel files with the extension "xls" or "xlsx" - for old or new format.

```
excel.SaveExcelFile(filename);
```

Use this member function of the **ExcelInterface** class to save an Excel file.

Definition:

```
void SaveExcelFile(string fileName)
```

Argument:

Variable	Function
fileName	The full path plus filename of the Excel file. Please save only files with the extension "xls" or "xlsx".

```
excel. AddExcelSheet(sheetName);
```

Use this member function of the **ExcelInterface** class to add a sheet to an Excel file.

Definition:

```
void AddExcelSheet(int row, int col, const std::string& stdExcelString)
```

Argument:

Variable	Function
sheetName	The sheet name.

```
excel.WriteStringToExcel(2, 0, "Hello Excel!"); // row, column, text
```

Use this member function of the **ExcelInterface** class to write a string to the (active) Excel sheet.

Definition:

```
void WriteStringToExcel(int row, int col, string excelString)
```

Argument:

Variable	Function
row	The row of your focus cell.
row	The column of your focus cell.
excelString	Your text string.

```
excel.WriteNumberToExcel(3, 1, 3.14); // row, column, double value
```

Use this member function of the **ExcelInterface** class to write a real value to the (active) Excel sheet.

Definition:

```
void WriteNumberToExcel(int row, int col, double value)
```

Argument:

Variable	Function
row	The row of your focus cell.
row	The column of your focus cell.
value	Your text value.

```
excel.ReleaseInterface(); // Important to do!
```

Use this member function of the **ExcelInterface** class to make a garbage collection and free the interface.

Definition:

```
void ReleaseInterface()
```

Hint
You should use this function!

That's it!

You can find this script file in the setup folder:

<..\Scriptings\ Write To Excel File.cpp>

Hint
Please have a look at the example "Write Text Labels to Excel File.cpp".

2.20 Import Excel Standard File

Objectives:

1. Load a sample evaluation
2. Get the chart object
3. Establish the Microsoft Excel Interface.
4. Import Excel File

With this script you can only import standard formatted Excel files. Standard means the format that you get when you export a table as an Excel file. The reason is, that there is yet no settings functionality available (Maybe in the next versions…) also not for the sheet name. It takes always the first sheet.

```
app.ImportExcelFile(ch, "e:\\test.xls", resetStyle, graphNameFromFirstRow,
                    axisNameFromFirstSecond);
```

Use this member function of the **Application** interface to import an Excel file.

Definition:

```
void ImportExcelFile(Chart ch, string fileName, bool resetStyle, bool graphNameFromFirstRow,
                     bool axisNameFromSecondRow)
```

Argument:

Variable	Function
Chart	Chart Object Use `app.GetChart()` to get it.
fileName	The full path plus filename of the Excel file. Please use only Excel files with the extension "xls" or "xlsx" - for old or new format.
resetStyle	If true then the style of the graphs will change to default (marker style, colors, line thickness, etc.)
graphNameFromFirstRow	Takes the name of each graph from first Excel row
axisNameFromSecondRow	Takes the name of each axis column from second Excel row

Here the example script to import a sheet from an Excel file.

```
#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetSimplexAppPath();
```

```
  string filename = simplexAppPath + "Examples\\MainPlots\\WinCC Sample
                                                                  Trend.sx";
  app.Output(filename);

  if (app.FileExist(filename))
  {
    // Load the evaluation
    app.LoadEval(filename);

    Chart ch = app.GetChart("MyChart");

    bool resetStyle = true;
    bool graphNameFromFirstRow = true;
    bool axisNameFromSecondRow = true;

    app.ImportExcelFile(ch, "e:\\test.xls", resetStyle, graphNameFromFirstRow,
                         axisNameFromSecondRow);
    ch.AutoScale();
  }
  else
  {
    app.Error("File <" + filename + "> does not exist!");
  }
}
```

(from ..\Scriptings\ImportExcelFile.cpp)

2.21 Make Surface Plot

Objectives:

1. Make a ***Surface Plot***

The following code is intended to make a surface plot with your own data (here generated via simple program). We have seen nearly all of the functions from this code in the previous chapters. Maybe the handle functions at the end of the main function are new for you, but they are straightforward and not complicated.

```
#pragma extension "corelib"

#define X_GRID_DENSITY 30
#define Y_GRID_DENSITY 30

double min(double a, double b)
{
  return ((a < b) ? a : b);
}

double max(double a, double b)
{
  return ((a > b) ? a : b);
}

void main()
{
  Application app("My App");
  app.NewEval();

  double Period1 = 2 * 3.141592654f / 4;
  double Amplitude1 = 0.25;
  double Amplitude2 = 1;
  double Period2 = 2 * 3.141592654f / 16;
  double Amplitude3 = 2;

  double xmin = -10.0;
  double xmax = 10.0;
  double dx = 2.0;
  double zmin = 0.0;
  double zmax = 3.0;
  double dz = 1.0;
  double ymin = -10.0;
  double ymax = 0.0;
  double dy = 2.0;

  array<float> ax(X_GRID_DENSITY);
  array<float> ay(Y_GRID_DENSITY);
  array<float> az(X_GRID_DENSITY * Y_GRID_DENSITY);

  for (int i = 0; i < X_GRID_DENSITY; i++)
```

```
  {
    double x = xmin + i * ((xmax - xmin) / (X_GRID_DENSITY - 1));
    ax[i] = x;
  }

  for (int j = 0; j < Y_GRID_DENSITY; j++)
  {
    double y = ymin + j * ((ymax - ymin) / (Y_GRID_DENSITY - 1));
    ay[j] = y;
  }

  for (int yIndex = 0; yIndex < Y_GRID_DENSITY; yIndex++)
  {
    double y = ax[yIndex];

    for (int xIndex = 0; xIndex < X_GRID_DENSITY; xIndex++)
    {
      double x = ay[xIndex];

      double Rd = GetRandomValue(0, 80);
      double z = Amplitude2 * cos(Period2 * x) + Amplitude3 * cos(Period1 * y) /
(abs(y) / 3.0 + 1) + Amplitude1 * Rd;

      z = min(zmax, z);
      z = max(zmin, z);

      az[yIndex * Y_GRID_DENSITY + xIndex] = z;
    }
  }

  //____________________________________________________________________________

  Chart ch = app.MakeChart("Surface Plot", idChartTypeSurface,100,100,400,300);
  //____________________________________________________________________________

  int m = X_GRID_DENSITY * Y_GRID_DENSITY;
  ch.SetNumberOfSampleData(m, 0);

  for (int yIndex = 0; yIndex < Y_GRID_DENSITY; yIndex++)
  {
    for (int xIndex = 0; xIndex < X_GRID_DENSITY; xIndex++)
    {
      int zIndex = yIndex * Y_GRID_DENSITY + xIndex;

      ch.SetDataX(zIndex, 0, ax[xIndex]);
      ch.SetDataY(zIndex, 0, ay[yIndex]);
      ch.SetDataZ(zIndex, 0, az[zIndex]);
    }
  }

  ch.SetSurfaceScatterPlot();

  ch.SetWidth(ch.GetWidth() * 2);
  ch.SetHeight(ch.GetHeight() * 1.1);
  ch.MoveTo(10, 10);
```

```
  ch.MoveLeftHandle(-123456); // To the smallest left border
  ch.MoveTopHandle(-50);
  ch.MoveRightHandle(1000);
  ch.MoveBottomHandle(150);

  app.SelectChart("Surface Plot");

  ch.AutoScale();
}
```

That's the last example!

You can find this script file in the setup folder:

`<..\Scriptings\ Make Surface Plot.cpp>`

Hint

If you have any questions related to these examples, please do not hesitate to contact our support.

2.22 Rotate 3D Surface Plot

Objectives:

1. Load any Evaluation
2. Get the chart object
3. Endless Loop
4. Rotate Surface Plot

There are three functions to use for rotating a surface plot:

Definition:

```
void SetRotationAngle(double angle);

void SetElevationAngle(double angle);

void SetTwistAngle(double angle);
```

Return Value:

nothing

Arguments:

angle	the corresponding angle

Next example will rotate a surface plot endless time on the screen (Exit with the Esc key).

```
#pragma extension "corelib"

#define ever (;;)

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetExamplesPath();

  string filename = simplexAppPath + "Animation/Eval/Surface Plot.sx";

  app.Output(filename);

  if (app.FileExist(filename))
  {
    app.LoadEval(filename);

    Chart ch = app.GetChart("SurfacePlot");
    ch.SetActiveGraph(0);

    int rot = 38, elev = 31, twist = 2;
    int Id = 1;

    ch.SetRotationAngle(rot);
    ch.SetElevationAngle(elev);
    ch.SetTwistAngle(twist);
```

```
    int counter = 0;

   for ever
   {
    if (app.EscapeLoop()) // Press key Esc and leave the loop
        break;

      app.DelayMS(100);

      if (!app.IsGraphicsViewAvailable()) // Do NOT forget this!
        break;

      ch.SetRotationAngle(rot);
      rot += 10;

      if (rot >= 360)
      {
        rot = 38;
        ch.SetElevationAngle(elev);
        elev += 50;
      }

      if (elev >= 360)
      {
        elev = 2;
        ch.SetTwistAngle(twist);
        twist += 50;
      }

      if (counter > 0 && counter % 100 == 0)
      {
        ch.SetSurfaceFunctionId(Id++);

        if (Id >= 23)
          Id = 1;
      }

      counter++;
    }
  }
  else
  {
    app.Error("File does not exist!");
  }
}
```

(from ..\Scriptings\Animate Surface Plot.cpp)

2.23 Database Import

Objectives:

1. Load any Evaluation
2. Make an instance of the database class
3. Connect to Database
4. Run Query
5. Save Query Results
6. Transfer to DataSheet
7. Release Interface

These are the steps to import data (trends) from a database. The following function names and its arguments should be self-explaining for a script programmer.

2.23.1 Make an instance of the database class

```
// Database Interface
DBInterface db("MyDB");
```

DBInterface class encapsulate the database interface and its methods (functions). You need to declare it first, before you can call any database related function.

2.23.2 Connect to Database

```
int type = DB_MS_ACCESS;
string ServerName = "MyServer2";  // Not used for Microsoft Access!
string DatabaseName = "MyDB";     // dto.
string LoginName = "MyName";      // dto.
string Password = "MyPW";         // dto.
string DatabaseFilename = dbPath; // Used for Microsoft Access, only!
string DatabaseSQLString = "SELECT TOP 100 * FROM [DataTable]";

db.ConnectToDatabase(type, ServerName, DatabaseName, LoginName, Password,
                        DatabaseFilename, DatabaseSQLString);
```

To connect to your database, the appropriated database provider has to be installed, before. And certainly the database has to be properly established. For details about driver installation and data source setup, please see corresponding documentation…

Definition:

```
bool ConnectToDatabase(string serverName, string DatabaseName, string LoginName,
                       string Password, string DatabaseFilename, string DatabaseSQLString)
```

Return Value:

`false` if something goes wrong; `true` if ok

Arguments:

(Self-explaining)

2.23.3 Run Query

```
if (!db.RunQuery())
   return;
```

The function *RunQuery* requests for information from the connected database. A query is an inquiry into the database unsing the SELECT statement **`DatabaseSQLString`** (see the last argument of `ConnectToDatabase`).

Definition:

```
bool RunQuery()
```

Return Value:

`false` if something goes wrong; `true` if ok

Arguments:

`nothing`

2.23.4 Save Query Results

```
// If you have forgotten this, SxN will do it the next time for you!
db.ReleaseInterface();
```

The function *SaveQueryResults* saves the query results (e.g. the the trend data) into a temporary Excel file. A typical path can be:

```
C:\Users\Ralf.MyPC\AppData\Local\Temp\sxg_query_db\SxGAE18.tmp
```

→Yes, you can rename this file in ‘`My Query Result.xls`’

Definition:

```
bool SaveQueryResults()
```

Return Value:

`false` if something goes wrong; `true` if ok

Arguments:

`nothing`

2.23.5 Transfer to DataSheet

```
bool resetStyle = true;
bool graphNameFromFirstRow = true;
bool axisNameFromSecondRow = true;

app.TransferToDataSheet(ch, resetStyle, graphNameFromFirstRow,
                        axisNameFromSecondRow);
```

The function *TransferToDataSheet* transfers the query results back from above temporary Excel file into *SimplexNumerica’s DataSheet*.

Definition:

```
bool TransferToDataSheet(Chart ch, bool resetStyle, bool graphNameFromFirstRow,
                         bool axisNameFromSecondRow)
```

Return Value:

`false` if something goes wrong; `true` if ok

Arguments:

`nothing`

2.23.6 Release Interface

```
// If you have forgotten this, SxN will do it the next time for you!
db.ReleaseInterface();
```

We need to release the instance of the ***DBInterface***.

Definition:

```
bool ReleaseInterface()
```

Return Value:

`nothing`

Arguments:

`nothing`

To see how it works in the script code, here the whole example script:

```
#pragma extension "corelib"

enum _tagDatabaseTypes
{
  DB_MS_ACCESS = 0,
  DB_MS_SQL_SERVER,
  DB_MYSQL,
  DB_IBM_DB2,
  DB_ORACLE_ORACLE_PROVIDER,
  DB_ORACLE_MS_PROVIDER,
  DB_ALL_PROVIDER,
  DB_WINCC_OLE_DB_PROVIDER,
};

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetSimplexAppPath();

  string filename = simplexAppPath + "Database\\Test DB.sx";
```

```
  string dbPath = simplexAppPath + "Database\\AMN Gas Engine.mdb";

  if (app.FileExist(filename))
  {
    // Load the evaluation
    app.LoadEval(filename);

    Chart ch = app.GetChart("My DBChart");

    // Database Interface
    DBInterface db("MyDB");

    int type = DB_MS_ACCESS;
    string ServerName = "MyServer2";  // Not used for Microsoft Access!
    string DatabaseName = "MyDB";     // dto.
    string LoginName = "MyName";      // dto.
    string Password = "MyPW";         // dto.
    string DatabaseFilename = dbPath; // Used for Microsoft Access, only!
    string DatabaseSQLString = "SELECT TOP 100 * FROM [DataTable]";

    if (!db.ConnectToDatabase(type, ServerName, DatabaseName, LoginName,
                              Password, DatabaseFilename, DatabaseSQLString))
      return;

    if (!db.RunQuery())
      return;

    if (!db.SaveQueryResults())
      return;

    bool resetStyle = true;
    bool graphNameFromFirstRow = true;
    bool axisNameFromSecondRow = true;

    app.TransferToDataSheet(ch, resetStyle, graphNameFromFirstRow,
                            axisNameFromSecondRow);

    db.ReleaseInterface(); // If you have forgotton this, SxN will..
  }
  else
  {
    app.Error("File <" + filename + "> does not exist!");
  }
}
```

(from ..\Scriptings\Database Import.cpp)

2.24 WinCC Database Import

SIMATIC WinCC is a SCADA System from Siemens. SCADA systems are used to monitor and control physical processes involved in industry and infrastructure on a large scale and over long distances. SIMATIC WinCC can be used in combination with Siemens controllers (PLCs).

Objectives:

1. Import WinCC Data Archive

The functions to import are similar to previous chapter, so that we'll only show the script code here:

```
/*****************************************************************
   Simplex - Open and Query a WinCC Archive SQL Server Database
*****************************************************************/

#pragma extension "corelib"

void main()
{
  Application app("Simple App");

  string simplexAppPath = app.GetSimplexAppPath();

  string filename = simplexAppPath + "Examples\\MainPlots\\WinCC Sample Trend.sx";
  app.Output(filename);

  if (app.FileExist(filename))
  {
    // Load the evaluation
    app.LoadEval(filename);

    Chart ch = app.GetChart("MyChart");

    // WinCC Database Interface
    WinCCDBInterface db("WinCC");

    string ServerName = "MyServer2";
    string DatabaseName = "MyDB";
    string LoginName = "MyName";
    string Password = "MyPW";

    if (!db.ConnectToDatabase(ServerName, DatabaseName, LoginName, Password))
      return;

    if (!db.RunQuery())
      return;

    if (!db.GetQueryResults("", ""))
      return;

    db.RemoveAllRequeryTags();
    db.AddRequeryTag("MyArchiveTag1", 0.1);
    db.AddRequeryTag("MyArchiveTag2", 0.1);
    db.AddRequeryTag("MyArchiveTag3", 0.1);

    bool useDateTimeFromChart = false;
    string startDateTime = "2017-12-09 09:03:00.000";
```

```
    string endDateTime = "2017-12-09 12:03:00.000";

    if (useDateTimeFromChart)
      app.GetDateTimeFromChart(ch, startDateTime, endDateTime);

    int nTimestep = 30; //sec.
    int nAggrSelected = 1;
    db.RunRequery(startDateTime, endDateTime, nTimestep, nAggrSelected);

    bool bUseScherenschnitt = true;
    bool bNormalizeData = true;
    int nMaxNumberToShow = 10000;

    if (!db.RecalcRequeryResults(bUseScherenschnitt, bNormalizeData,
                                 nMaxNumberToShow))
      return;

    if (!db.PrepareForDataSheetFormat())
      return;

    bool resetStyle = true;
    bool graphNameFromFirstRow = true;
    bool axisNameFromFirstSecond = true;

    app.TransferToDataSheet(ch, resetStyle, graphNameFromFirstRow,
                            axisNameFromFirstSecond);
    ch.AutoScale();

    db.ReleaseInterface(); // If you have forgotton this, SxN will do it!
  }
  else
  {
    app.Error("File <" + filename + "> does not exist!");
  }
}
```

(from ..\Scriptings\WinCC Database Import.cpp)

2.25 Spreadsheet Base Functions

The new SimplexNumerica Excel-like Spreadsheet is associated to some base script functions.

Objectives:

1. Spreadsheet Base Functions

Please have a look to the example file:

```
..\Scriptings\Spreadsheet Script.cpp
```

```
app.NewSpreadsheet();
```

Use this short member function of the ***Application*** class to make a new spreadsheet window.

Definition:

```
void NewSpreadsheet()
```

```
app.ActivateSheet("Sheet1");
```

Use this member function of the **Application** interface to activate the sheet inside the spreadsheet.

Definition:

```
void ActivateSheet(string sheetName)
```

Argument:

Variable	Function
String sheetName	The name of the sheet

```
Sheet mySheet = app.GetActiveSheet();
```

Use this member function of the ***Application*** class to get a reference to the current active sheet.

Definition:

```
Sheet GetActiveSheet()
```

```
app.NewSheet("Sheet2"); or use
app.AddSheet("Sheet2");
```

Use one of this member function of the **Application** interface to make a new sheet and activate the sheet inside the spreadsheet.

Definition:

```
void NewSheet(string sheetName)
```

Argument:

Variable	Function
string sheetName	The name of the new sheet

```
app.RemoveSheet("Sheet1");
```

Use this member function of the **Application** interface to remove (delete) an existing sheet and activate one of the other sheets inside the spreadsheet.

Definition:

```
void RemoveSheet(string sheetName)
```

Argument:

Variable	Function
string sheetName	The name of the sheet to remove

```
app.RenameSheet("Sheet1", "SheetX");
```

Use this member function of the **Application** interface to rename an existing sheet.

Definition:

```
void RenameSheet(string sheetNameFrom, string sheetNameTo)
```

Argument:

Variable	Function
string sheetNameFrom	The original name of the sheet
string sheetNameTo	The new name of the sheet

```
app.LoadSpreadsheet("c:/Test/MySpreadsheet.sxl"); or
app.LoadSpreadsheet("c:\\Test\\MySpreadsheet.sxl");
```

Use this member function of the **Application** interface to load a new spreadsheet from disk.

Definition:

```
void LoadSpreadsheet(string fileName)
```

Argument:

Variable	Function
string fileName	The file name of the spreadsheet

```
app.SaveSpreadsheet("c:/Test/MySpreadsheet.sxl"); or
app.SaveSpreadsheet("c:\\Test\\MySpreadsheet.sxl");
```

Use this member function of the **Application** interface to save a spreadsheet to disk.

Definition:

```
void SaveSpreadsheet(string fileName)
```

Argument:

Variable	Function
string fileName	The file name of the spreadsheet

Here is he sample script from the file `..\Scriptings\Spreadsheet Script.cpp`

```
/*********************************************
   Simplex - Default Script
*********************************************/

#pragma extension "corelib"

// #define LOAD_A_NEW_SHEET

// Cell Main Formats
#define FORMAT_AUTOMATIC 0 // Automatic
#define FORMAT_COUNTRY_SPEC 1 // Standard
#define FORMAT_SCIENTIFIC 2 // Scientific
#define FORMAT_ENGINEERING 3 // Engineering
#define FORMAT_TECHNICAL 4 // Technical
#define FORMAT_DATE 5 // Date
#define FORMAT_TIME 6 // Time
#define FORMAT_DATETIME 7 // Date/Time
#define FORMAT_TIME_MILLISECOND 8 // Time plus msec
#define FORMAT_DATETIME_MILLISECOND 9 // Date/Time plus msec
#define FORMAT_GENERIC_STRING 10 // Generic String
#define FORMAT_WORDS 11 // Numbers in Words
#define FORMAT_DATETIME_EXTRA 12 // Extra Date/Time Format
#define FORMAT_DATETIME_MILLISECOND_EXTRA 13 // Extra Date/Time plus msec
#define FORMAT_TEXT_STRING 14 // Text String

// Cell Extra Date/Time Format Index
// see function below <int GetExtraDateTimeIndex(int format)>

void main()
{
  Application app("My Workbook");

  // Make a new spreadsheet workbook with three default sheets
  app.NewSpreadsheet();

  // Play around the workbook
  app.ActivateSheet("Sheet2");
  app.NewSheet("My New Sheet"); // or use AddSheet
```

```
  app.RemoveSheet("Sheet3");
  app.RenameSheet("Sheet2", "My Table");

  // Load another spreadsheet from disk
#ifdef LOAD_A_NEW_SHEET
  const string filename = "E:/Sx/SxN-
2015/SxN64.2015/Tutorial/Spreadsheet/ExcelSheet.sxl";
  app.LoadSpreadsheet(filename);
  app.ActivateSheet("Sheet2");
#endif

  Sheet mySheet = app.GetActiveSheet();

  string sheetName = mySheet.GetName();
  app.Output("The active Sheet Name is: " + sheetName);

  // *** Switch off Undo and Redrawing!
  mySheet.BlockRedraw(); // Makes really only sense with much more cells!

  mySheet.SetCell(1, 1, sheetName); // Enter everything
  mySheet.SetFormat(1, 1, FORMAT_TEXT_STRING, 0);
  mySheet.SetFontName(1, 1, "Times New Roman");
  mySheet.SetFontSize(1, 1, 18);

  mySheet.SetCellValue(2, 1, sheetName); // Enter a string
  mySheet.SetFormat(2, 1, FORMAT_TEXT_STRING, 0);
  mySheet.SetFontBold(1, 1, true);

  double pi = 3.14;
  mySheet.SetCellValue(3, 1, pi / 10);    // Enter a number (double)
  mySheet.SetFormat(3, 1, FORMAT_COUNTRY_SPEC, 0);
  mySheet.SetDecimalPlaces(3, 1, 3);
  mySheet.SetTextColor(3, 1, RGB(255,0,0));
  mySheet.SetFontItalic(3, 1, true);

  mySheet.SetCellFormula(4, 1, "sin(A3)/A3");  // Enter a formula
  mySheet.SetFormat(4, 1, FORMAT_COUNTRY_SPEC, 0);
  mySheet.SetDecimalPlaces(4, 1, 6);
  mySheet.SetCellColor(4, 1, RGB(0,255,200));
  mySheet.SetFontUnderline(4, 1, true);
  mySheet.SetFontItalic(4, 1, false);

  string cellValue = mySheet.GetCell(1, 1);
  double realValue = mySheet.GetCellValue(3, 1);
  string cellFormula = mySheet.GetCellFormula(4, 1);
  string cellFormulaResult1 = mySheet.GetCellValue(4, 1);

  mySheet.SetCellValue(3, 1, pi);
  mySheet.Recalculate();
  double cellFormulaResult2 = mySheet.GetCellValue(4, 1);

  // *** If you set BlockRedraw(), then do not forget this!
  mySheet.Redraw(); // Switch on Undo and redraw the whole grid

  app.SaveSpreadsheet("E:/My Test Spreadsheet.sxl");
```

```
  // Output to the Output Window
  app.Output("Text in cell A1 = " + cellValue);
  app.Output("Value in cell A3 (set to pi/10) = " + realValue);
  app.Output("Formula  in cell A4 " + cellFormula); // incl. = sign
  app.Output("Formula result  in cell A4 (calc with pi/10) = " +
cellFormulaResult1); // incl. = sign
  app.Output("Formula result  in cell A4 (recalc with pi) = " +
cellFormulaResult2);
}

int GetExtraDateTimeIndex(int format, const string& extraDateTimeFormat)
{
  int index = 0;

  if (format == FORMAT_DATETIME_EXTRA)
  {
    if (extraDateTimeFormat == "Microsoft Windows Settings") index = 0;
    else if (extraDateTimeFormat == "DD.MM.YYYY hh:mm:ss") index = 1;
    else if (extraDateTimeFormat == "YYYYMMDD") index = 2;
    else if (extraDateTimeFormat == "YYYYDDMM") index = 3;
    else if (extraDateTimeFormat == "YYYY/MM/DD") index = 4;
    else if (extraDateTimeFormat == "YYYY/DD/MM") index = 5;
    else if (extraDateTimeFormat == "DD/MM/YYYY") index = 6;
    else if (extraDateTimeFormat == "MM/DD/YYYY") index = 7;
    else if (extraDateTimeFormat == "YYYY-MM-DD") index = 8;
    else if (extraDateTimeFormat == "YYYY-DD-MM") index = 9;
    else if (extraDateTimeFormat == "DD-MM-YYYY") index = 10;
    else if (extraDateTimeFormat == "MM-DD-YYYY") index = 11;
    else if (extraDateTimeFormat == "DD.MM.YYYY") index = 12;
    else if (extraDateTimeFormat == "MM.DD.YYYY") index = 13;
    else if (extraDateTimeFormat == "DD-Mon-YY") index = 14;
    else if (extraDateTimeFormat == "DD-Mon-YY") index = 15;
    else if (extraDateTimeFormat == "DD-Mon-YYYY") index = 16;
    else if (extraDateTimeFormat == "DD-Mon-YYYY") index = 17;
    else if (extraDateTimeFormat == "hh:mm:ss") index = 18;
    else if (extraDateTimeFormat == "hh mm ss") index = 19;
    else if (extraDateTimeFormat == "HH.MM.SS") index = 20;
    else if (extraDateTimeFormat == "YYYYMMDD hh:mm:ss") index = 21;
    else if (extraDateTimeFormat == "YYYYDDMM hh:mm:ss") index = 22;
    else if (extraDateTimeFormat == "YYYY/MM/DD hh:mm:ss") index = 23;
    else if (extraDateTimeFormat == "YYYY/DD/MM hh:mm:ss") index = 24;
    else if (extraDateTimeFormat == "DD/MM/YYYY hh:mm:ss") index = 25;
    else if (extraDateTimeFormat == "MM/DD/YYYY hh:mm:ss") index = 26;
    else if (extraDateTimeFormat == "YYYY-MM-DD hh:mm:ss") index = 27;
    else if (extraDateTimeFormat == "YYYY-DD-MM hh:mm:ss") index = 28;
    else if (extraDateTimeFormat == "DD-MM-YYYY hh:mm:ss") index = 29;
    else if (extraDateTimeFormat == "MM-DD-YYYY hh:mm:ss") index = 30;
    else if (extraDateTimeFormat == "YYYY-MM-DDThh:mm:ss") index = 31;
    else if (extraDateTimeFormat == "YYYYMMDDThh:mm:ss") index = 32;
    else if (extraDateTimeFormat == "DD-MM-YYYYThh:mm:ss") index = 33;
    else if (extraDateTimeFormat == "YYYYMMDDTHHMM") index = 34;
    else if (extraDateTimeFormat == "ISO8601-0 (DateThh:mm:ssZ)") index = 35;
    else if (extraDateTimeFormat == "ISO8601-1 (DateThh:mm:ssTZD)") index = 36;
    else if (extraDateTimeFormat == "ISO8601-0 (DateThh:mmZ)") index = 37;
```

```
    else if (extraDateTimeFormat == "ISO8601-1 (DateThh:mmTZD)") index = 38;
    else if (extraDateTimeFormat == "RFC-822 HTTP DateTime") index = 39;
  }
  else if (format == FORMAT_DATETIME_MILLISECOND_EXTRA)
  {
    if (extraDateTimeFormat == "hh:mm:ss.mss") index = 0;
    else if (extraDateTimeFormat == "hh mm ss mss") index = 1;
    else if (extraDateTimeFormat == "HH.MM.SS.mss") index = 2;
    else if (extraDateTimeFormat == "YYYYMMDD hh:mm:ss.mss") index = 3;
    else if (extraDateTimeFormat == "YYYYDDMM hh:mm:ss.mss") index = 4;
    else if (extraDateTimeFormat == "YYYY/MM/DD hh:mm:ss.mss") index = 5;
    else if (extraDateTimeFormat == "YYYY/DD/MM hh:mm:ss.mss") index = 6;
    else if (extraDateTimeFormat == "DD/MM/YYYY hh:mm:ss.mss") index = 7;
    else if (extraDateTimeFormat == "MM/DD/YYYY hh:mm:ss.mss") index = 8;
    else if (extraDateTimeFormat == "YYYY-MM-DD hh:mm:ss.mss") index = 9;
    else if (extraDateTimeFormat == "YYYY-DD-MM hh:mm:ss.mss") index = 10;
    else if (extraDateTimeFormat == "DD-MM-YYYY hh:mm:ss.mss") index = 11;
    else if (extraDateTimeFormat == "MM-DD-YYYY hh:mm:ss.mss") index = 12;
    else if (extraDateTimeFormat == "YYYY-MM-DDThh:mm:ss.mss") index = 13;
    else if (extraDateTimeFormat == "YYYYMMDDThh:mm:ss.mss") index = 14;
    else if (extraDateTimeFormat == "DD-MM-YYYYThh:mm:ss.mss") index = 15;
    else if (extraDateTimeFormat == "YYYYMMDDTHHMmss") index = 16;
    else if (extraDateTimeFormat == "YYYYMMDDTHHMmssmss") index = 17;
    else if (extraDateTimeFormat == "NCSA Common Log DateTime") index = 18;
  }

  return index;
}
```

Now, we will explain the cell functions. Important here is the knowledge of the script function GetActiveSheet() as described above and here repeated again:

```
Sheet mySheet = app.GetActiveSheet();
```

Use this member function of the ***Application*** class to get a reference to the current active sheet.

Definition:

```
Sheet GetActiveSheet()
```

```
string sheetName = mySheet.GetName();
```

Use this member function of the ***Sheet*** interface to get the name of the sheet.

Definition:

```
string GetName()
```

Argument:

Variable	Function
[Return] string	The name of the sheet

```
string sheetName = mySheet.GetName();
```

Use this member function of the ***Sheet*** interface to get the name of the sheet.

Definition:

```
string GetName()
```

Argument:

Variable	Function
[Return] string	The name of the sheet

```
string cellContent = mySheet.GetCell(1, 1);
```

Use this member function of the ***Sheet*** interface to get the formatted content of a cell.

Definition:

```
string GetCell(int row, int col)
```

Argument:

Variable	Function
[Return] string	The content of a cell
int row	The row index
int col	The column index

```
string cellText = mySheet.GetCellString(1, 1);
```

Use this member function of the ***Sheet*** interface to get explicitly the string content of a cell. The content will not be formatted!

Definition:

```
string GetCellString(int row, int col)
```

Argument:

Variable	Function
[Return] string	The content of a cell
int row	The row index
int col	The column index

```
string cellFormula = mySheet.GetCellFormula(1, 1);
```

Use this member function of the ***Sheet*** interface to get explicitly the formula of a cell.

Definition:

```
string GetCellFormula(int row, int col)
```

Argument:

Variable	Function
[Return] string	The formula of a cell
int row	The row index
int col	The column index

```
double cellValue = mySheet.GetCellValue(1, 1);
```

Use this member function of the ***Sheet*** interface to get explicitly the formula of a cell.

Definition:

```
double GetCellValue(int row, int col)
```

Argument:

Variable	Function
[Return] double	The data value of a cell
int row	The row index
int col	The column index

```
mySheet.SetCell(1, 1, "3.14"); or

mySheet.SetCell(1, 1, "sin(0.1)/0.1");
```

Use this member function of the ***Sheet*** interface to set the content of a cell. The program decides what it is, a formula, string or a value.

Definition:

```
void SetCell(int row, int col, string content)
```

Argument:

Variable	Function
string content	The content for the cell
int row	The row index
int col	The column index

```
mySheet.SetCellFormula(1, 1, "sin(B2)/B2");
```

Use this member function of the ***Sheet*** interface to set a formula to a cell.

Definition:

```
void SetCellFormula(int row, int col, string formula)
```

Argument:

Variable	Function
string formula	The formula for the cell
int row	The row index
int col	The column index

```
mySheet.SetCellValue(2, 1, mySirName); // Enter a string or

double pi = 3.14;
mySheet.SetCellValue(3, 1, pi / 10);    // Enter a number (double)
```

Use this member function of the ***Sheet*** interface to set a value/string to a cell.

Definition:

```
void SetCellValue(int row, int col, double dValue)

void SetCellValue(int row, int col, string strValue)
```

Argument:

Variable	Function
string /double	The value for the cell
int row	The row index
int col	The column index

```
mySheet.SetFormat(3, 1, FORMAT_COUNTRY_SPEC, 0);
```

Use this member function of the ***Sheet*** interface to set a format to a cell.

Definition:

```
void SetFormat(int row, int col, uint format, uint dateTimeFormatIndex)
```

Argument:

Variable	Function

uint format	The format for the cell
	(see sample program above)
	`// Cell Main Formats`
	`#define FORMAT_AUTOMATIC 0 // Automatic`
	`#define FORMAT_COUNTRY_SPEC 1 // Standard`
	`#define FORMAT_SCIENTIFIC 2 // Scientific`
	`#define FORMAT_ENGINEERING 3 // Engineering`
	`#define FORMAT_TECHNICAL 4 // Technical`
	`#define FORMAT_DATE 5 // Date`
	`#define FORMAT_TIME 6 // Time`
	`#define FORMAT_DATETIME 7 // Date/Time`
	`#define FORMAT_TIME_MILLISECOND 8 // Time plus msec`
	`#define FORMAT_DATETIME_MILLISECOND 9 // Date/Time plus msec`
	`#define FORMAT_GENERIC_STRING 10 // Generic String`
	`#define FORMAT_WORDS 11 // Numbers in Words`
	`#define FORMAT_DATETIME_EXTRA 12 // Extra Date/Time Format`
	`#define FORMAT_DATETIME_MILLISECOND_EXTRA 13 // Extra Date/Time plus msec`
	`#define FORMAT_TEXT_STRING 14 // Text String`

uint dateTimeFormatIndex	The index for the foreign data/time format. see sample program above the function: `int GetExtraDateTimeIndex(int format, const string& extraDateTimeFormat)`
int row	The row index
int col	The column index

```
mySheet.SetDecimalPlaces(3, 1, 3);
```

Use this member function of the ***Sheet*** interface to set a decimal places of a cell.

Definition:

```
void SetDecimalPlaces(int row, int col, uint decimalPlaces)
```

Argument:

Variable	Function
uint decimalPlaces	The decimal places of the cell
int row	The row index
int col	The column index

```
mySheet.SetTextColor(3, 1, RGB(255,0,0));
```

Use this member function of the ***Sheet*** interface to set the text color of a cell.

Definition:

```
void SetTextColor(int row, int col, uint textColor)
```

Argument:

Variable	Function
uint textColor	The text color of the cell
int row	The row index
int col	The column index

```
mySheet.SetCellColor(3, 1, RGB(255,255,0));
```

Use this member function of the ***Sheet*** interface to set the cell color of a cell.

Definition:

```
void SetCellColor(int row, int col, uint textColor)
```

Argument:

Variable	Function
uint cellColor	The cell color of the cell
int row	The row index
int col	The column index

```
mySheet.SetFontName(1, 1, "Times New Roman");
mySheet.SetFontSize(1, 1, 18);
mySheet.SetFormat(2, 1, FORMAT_TEXT_STRING, 0);
mySheet.SetFontBold(1, 1, true);
```

Use this member function of the ***Sheet*** interface to set the font of a cell.

Definition:

```
void SetFontName(int nRow, int nCol, const string stdFontname);

void SetFontSize(int nRow, int nCol, int nSize);

void SetFontBold(int nRow, int nCol, bool bBold);

void SetFontItalic(int nRow, int nCol, bool bItalic);

void SetFontUnderline(int nRow, int nCol, bool bUnderline);
```

Argument:

Variable	Function
[Arguments]	see yourself
int nRow	The row index
int nCol	The column index

```
mySheet.BlockRedraw();
```

Use this member function of the **Application** interface to block an existing sheet from undo and redrawing each time. Much faster in that way. See above example!

Definition:

```
void BlockRedraw()
```

Argument:

Variable	Function
-	-

```
mySheet.Redraw();
```

Use this member function of the **Application** interface to redraw the content of a sheet.

Definition:

```
void Redraw()
```

Argument:

Variable	Function
-	-

```
mySheet.Recalc();
```

Use this member function of the **Application** interface to recalc a whole sheet. Each formula of a cell (if available) will be re-calculated!

Definition:

```
void Recalc()
```

Argument:

Variable	Function
-	-

---

3 Call Script from Button

SimplexNumerica V14.1 introduces the functionality to call a script from a (button) shape, placed in the *Graphics View*.

Objectives:

1. Click on a button and call a script with different methods:

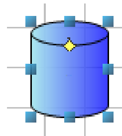
   Method 1: Script with the C++ main() function.

   Method 2: Script with any desired C++ function with double declared arguments and a double declared return value. Method 2 has two variations:
   Approach I: The clicked on shape gets the return value!
   Approach II: Another shape gets the return value!

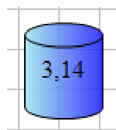
Before we come to the two methods, a short excurse on how to make any geometrical shape to a text shape.

3.1 Make a shape to a text shape

This is a normal shape (without text):

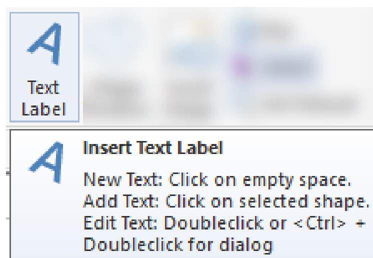


To transfer it to a text shape like



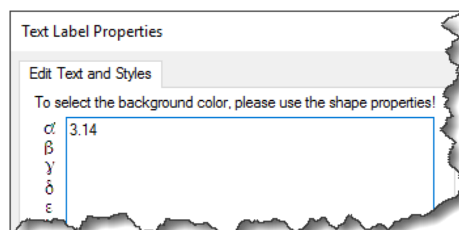
use the Ribbonbar *Edit* icon *Text Label*

(read the tooltip)



➔ Select a shape, click on that *Text Label* icon and place the mouse cursor over the selected shape, then press left mouse button to open the following dialogbox.

Edit the text label – here the value 3.14 or 3,14.



2. Now we have a new text label!

3.2 Method 1: Script with a main() function

➔ Click on a button and call the main script!

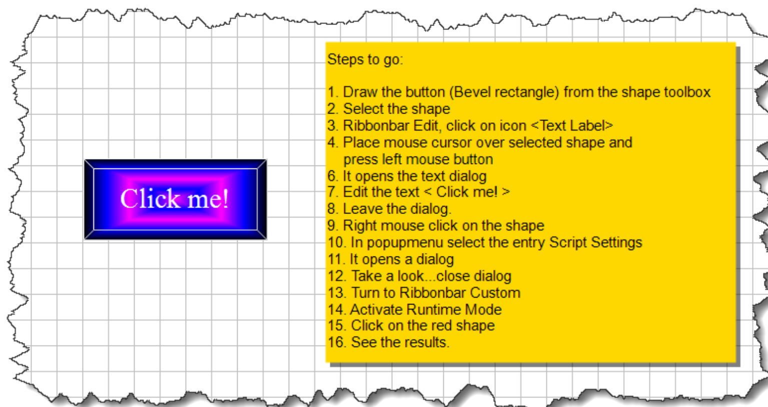
Have a look to the example evaluation in the scripting path:

```
..\Examples\Scripting\Hello World.sx
```

and the associated script in the folder

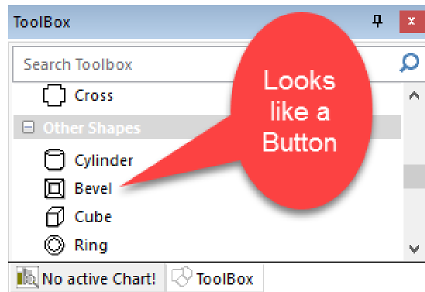
```
..\Scriptings\Hello World.cpp
```

The example evaluation <`Hello World.sx`> displays one single shape and a legend on the right side.

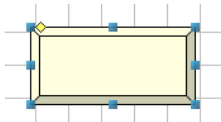


Follow the steps:

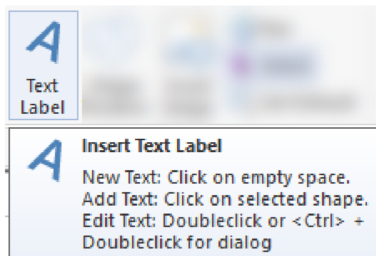
1. Draw the button (Bevel rectangle) from the shape toolbox and change the look…



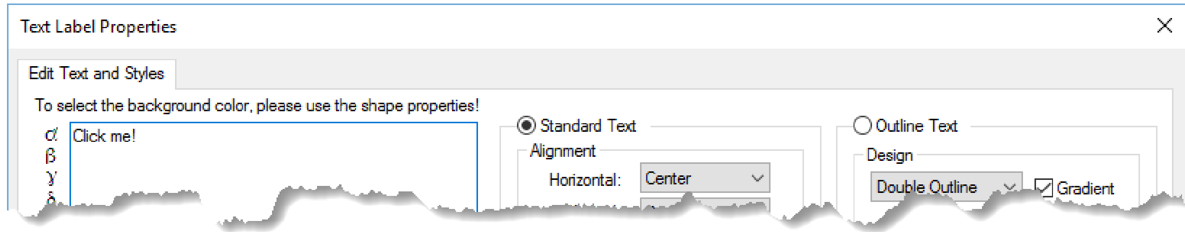
2. Select the button shape



3. Go to *Ribbonbar Edit*, click on icon <Text Label>

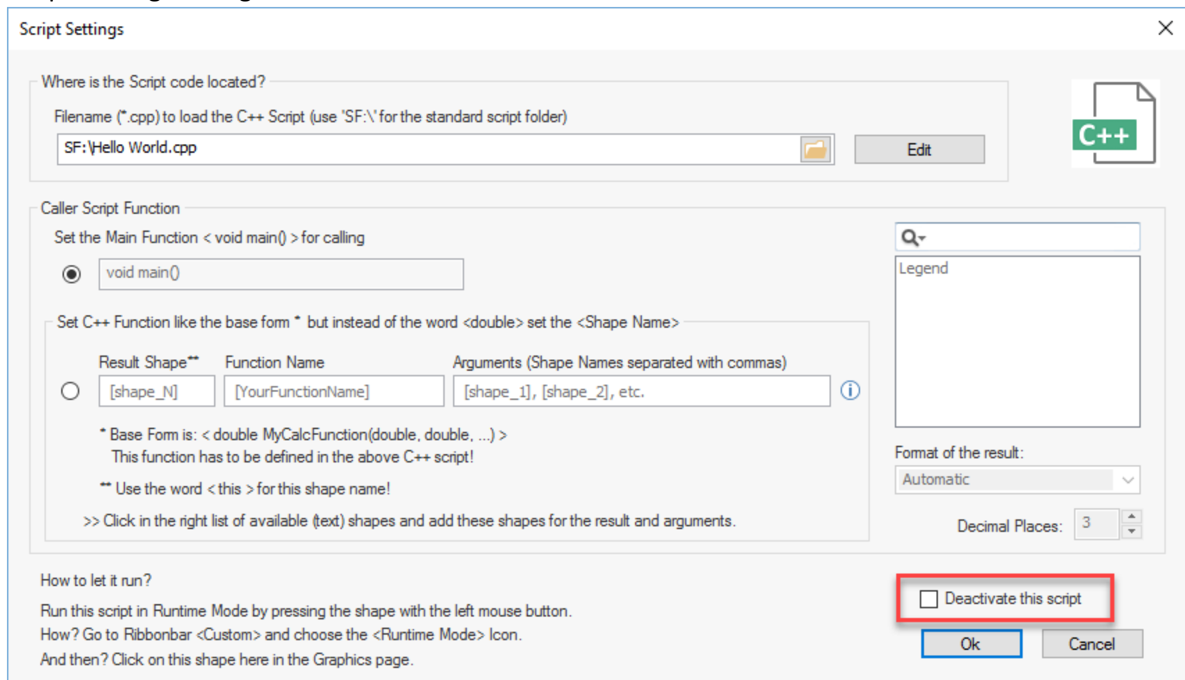


4. Place mouse cursor over selected shape and press left mouse button - that opens the text dialog:



5. Edit the text < Click me! > and leave the dialog with Ok.

6. Right mouse click on the shape and choose in Popupmenu the entry ‘Script Settings’. That opens the script settings dialog:



→Please set the file name as shown in the dialog and uncheck the red frame.

The often long standard path to the users scripting folder, like

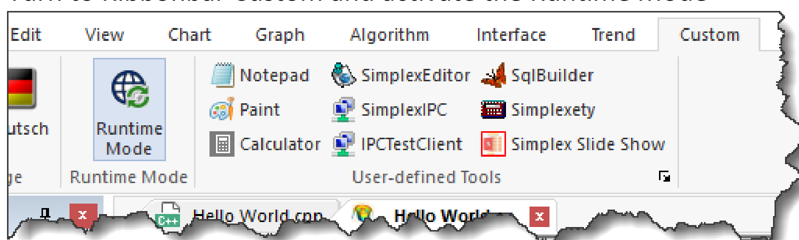
```
C:\Users\Ralf.P3\Documents\SimplexNumerica\Scriptings
```

can be abbreviated to

```
SF:\
```

7. Take a look around the dialog and close the dialog with Ok; there is nothing more to set, yet.

8. Turn to Ribbonbar Custom and activate the Runtime Mode



9. →Now, click on the red shape with the left mouse button!

10. The script code from the file <`Hello World.cpp`> will be called, immediately.

11. Follow the steps (or use copy & paste) for other (text) shapes.

3.3 Method 2: Script with any C++ function

➔ Click on a button and call a (mathematical) C++ function!

The mathematical oriented C++ script function must have the following syntax form with any desired names, like

```
double MyMathFunction(double dPower, double dFrequency, etc.)
```

but the type is always double precision!

Have a look to the example evaluation in the scripting path:

```
..\Examples\Scripting\Add.sx
```

and the associated script in the folder

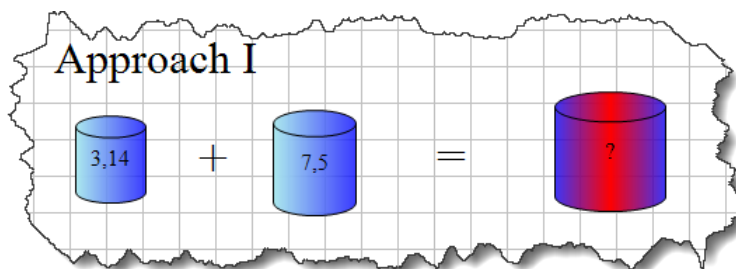
```
..\Scriptings\Calc.cpp
```

This example evaluation shows some shapes placed on the graphics page separated in two groups (Approach 1 + 2). The difference is the 'return value' to be placed on which shape's text (Property Shape Text); either the on the just before clicked shape (this) or another text shape (see 'Script Settings' → right click on a text shape).

3.3.1 Approach I

➔ The clicked on shape gets the return value!

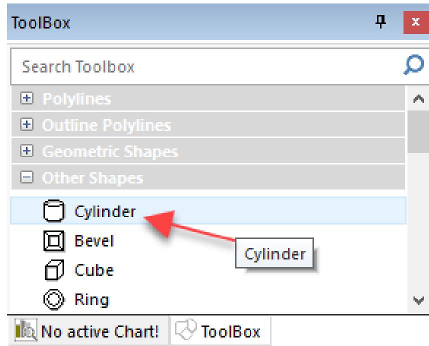
Next picture is copied from `..\Examples\Scripting\Add.sx`



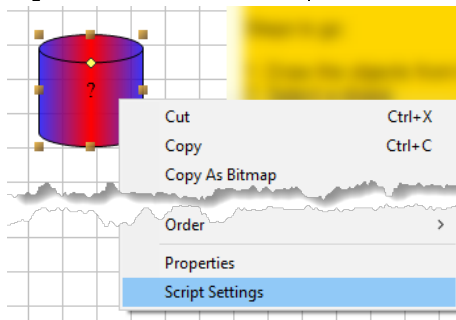
The red shape should earn the result when you click on it!

<u>Steps to go:</u>

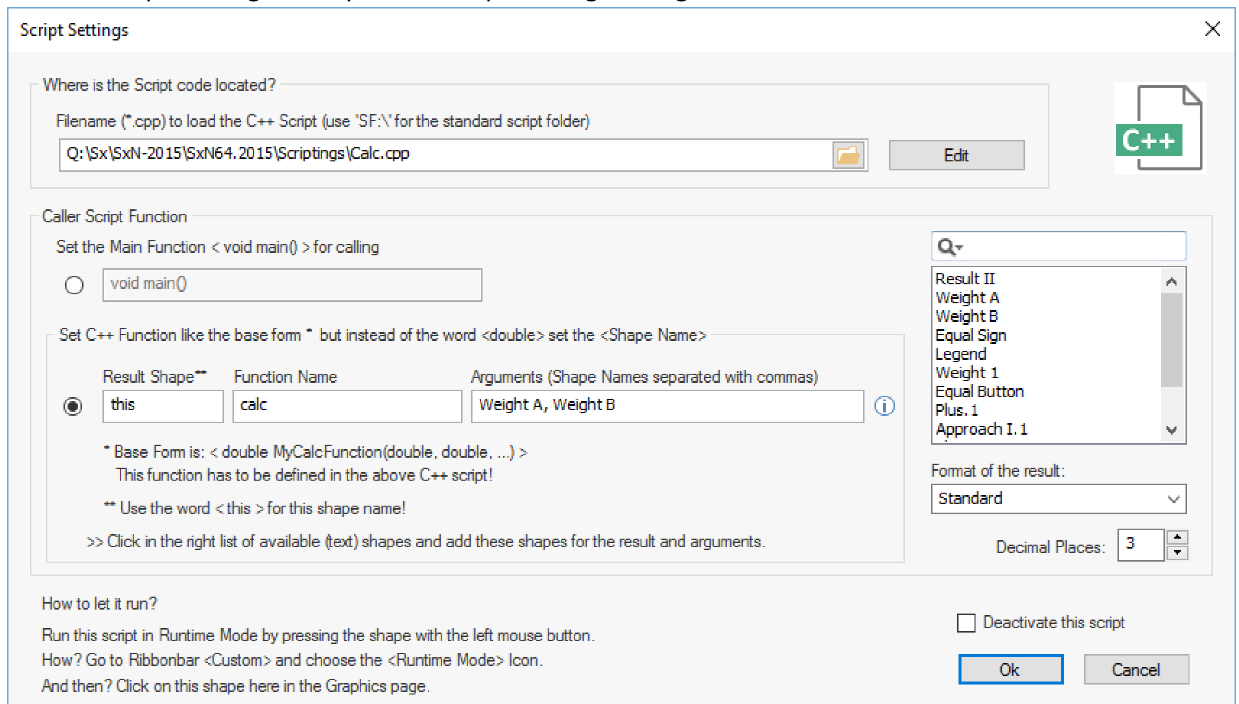
1. Draw the objects from the shape toolbox (here the cylinder shape) and modify the look…



2. Right click on the red shape



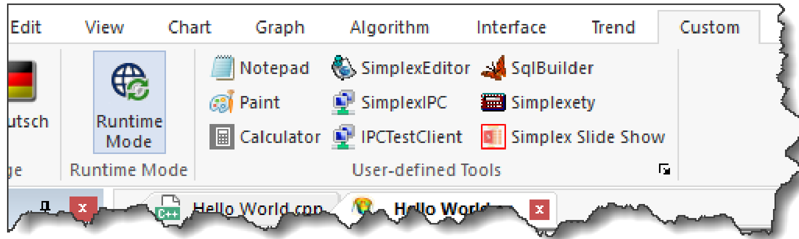
3. Choose Script Settings and open the script settings dialog.



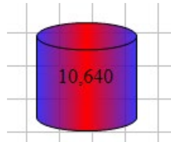
→Set the *Result Shape* to ***this***, *Function Name* and the *Arguments* as shown.
→Use the right list box and double-click on an entry.
→Format the result value.

4. Take a look around the dialog and close the dialog with Ok; Before, activate the script (uncheck the checkbox).

5. Turn to Ribbonbar Custom and activate the Runtime Mode



6. →Now, click on the red shape with the left mouse button!



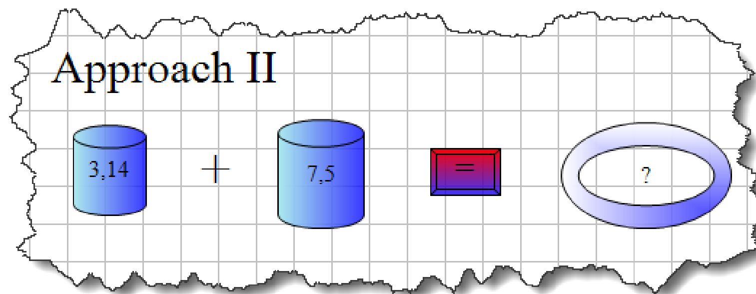
7. The script code from the file <`Calc.cpp`> will be called, immediately.

8. The result is set to the calling shape.

3.3.2 Approach II

➔ Another shape gets the return value!

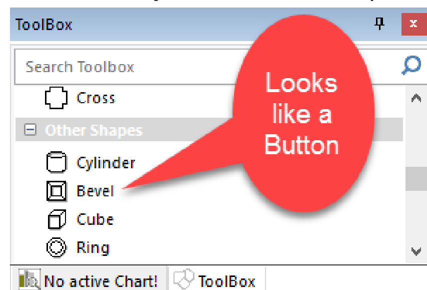
Next picture is also copied from `..\Examples\Scripting\Add.sx`



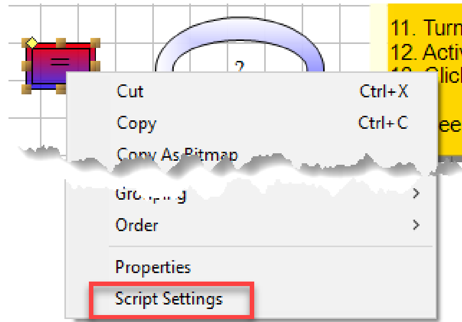
The right ellipse shape should earn the result when you click on the red one!

Steps to go:

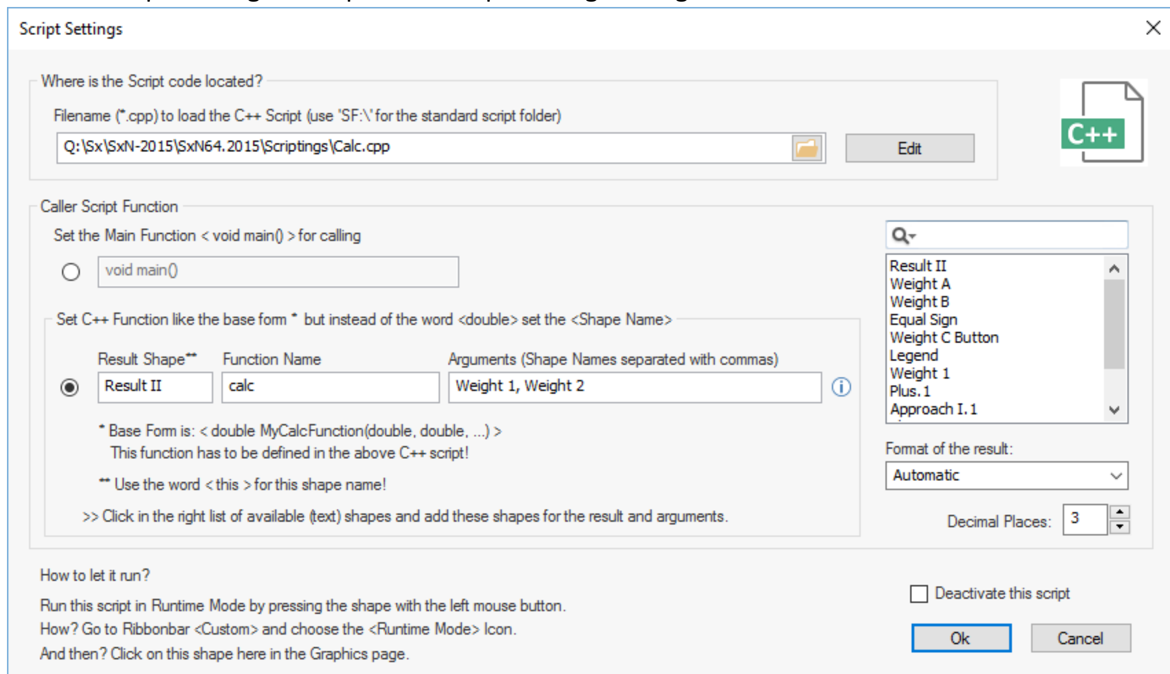
9. Draw the objects from the shape toolbox (here the cylinder shape) and modify the look…



10. Right click on the red equal (=) shape



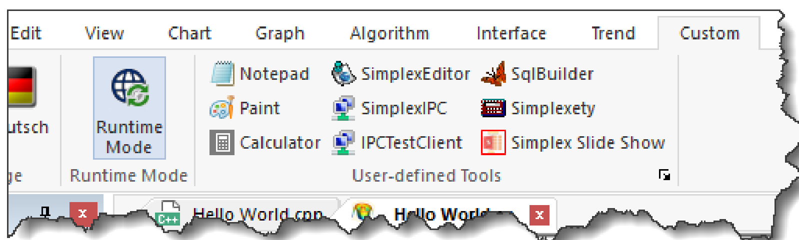
11. Choose Script Settings and open the script settings dialog.



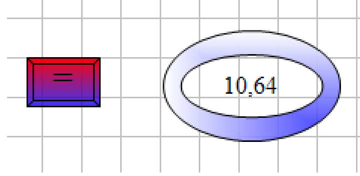
→Set the <Result Shape>, <Function Name> and the <Arguments> as shown.
→Use the right list box and double-click on an entry.
→Format the result value.

12. Take a look around the dialog and close the dialog with Ok; Before, activate the script (uncheck the checkbox).

13. Turn to Ribbonbar Custom and activate the Runtime Mode



14. →Now, click on the equal shape with the left mouse button!



15. The script code from the file <`Calc.cpp`> will be called, immediately.

16. The result is set to the other shape.

4 Simplex Remote Control (SimplexIPC)

This is an external application to control ***SimplexNumerica*** remotely from outside. The app is called ***Simplex Remote Control*** or short ***SimplexIPC*** *(Simplex **I**nter **P**rocess **C**ontrol)*.

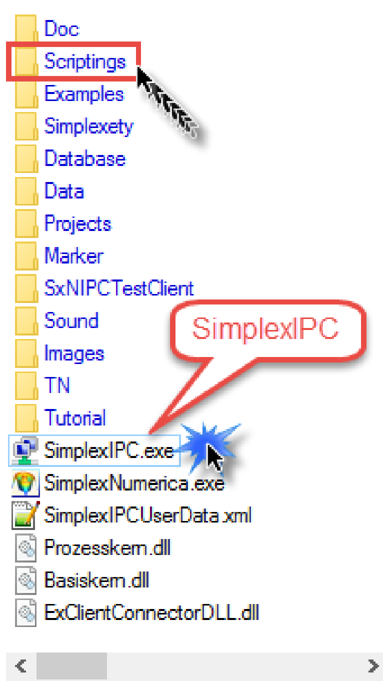
Use *SimplexIPC* to write different C++ scripts in *AngelScript* in the same manner as you would do it inside *SimplexNumerica*. Then connect to any *SimplexNumerica* running PC in the LAN IP range and send the script, so that SimplexNumerica on the other PC can compile and run it (certainly you can do it also on the same PC).

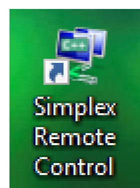
What is the purpose for this?

To answer the purposes, here some points of interest:

- Control *SimplexNumerica* from an external PC.
- Control different PCs from one station.
- Evaluate different measurement experiments at plant side and store the evaluation file on the LAN, only (and not all or the raw data rows).
- Let run standard scripts on a daily base (even perhaps on more stations).
- Help other users with executions of any sequences.
- Demonstrate from outside or make a presentation…
- etc.

You can see, based on these bullet points, that it can make sense to use such a tool like *SimplexIPC* to remote control *SimplexNumerica*.





Please start *SimplexIPC* from desktop shortcut or direct from Windows Explorer. Then look for the installation path to find the executables.

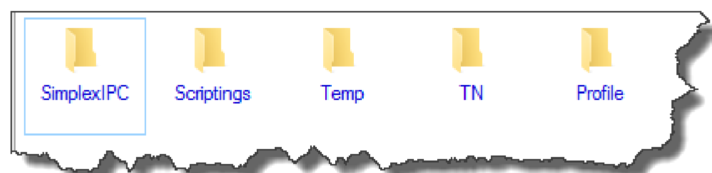
As you can see on the left picture, there is also the ***Original*** folder for the (sample) scriptings available.

As often discussed, if a user does not have ***User Rights*** for the installation folder (or gets trouble with administrator rights), then neither the user nor the program itself can write or modify files in this setup folder.

As a consequence of this (often unnecessary MS Windows behavior) we have to use your user directory, like personally for me

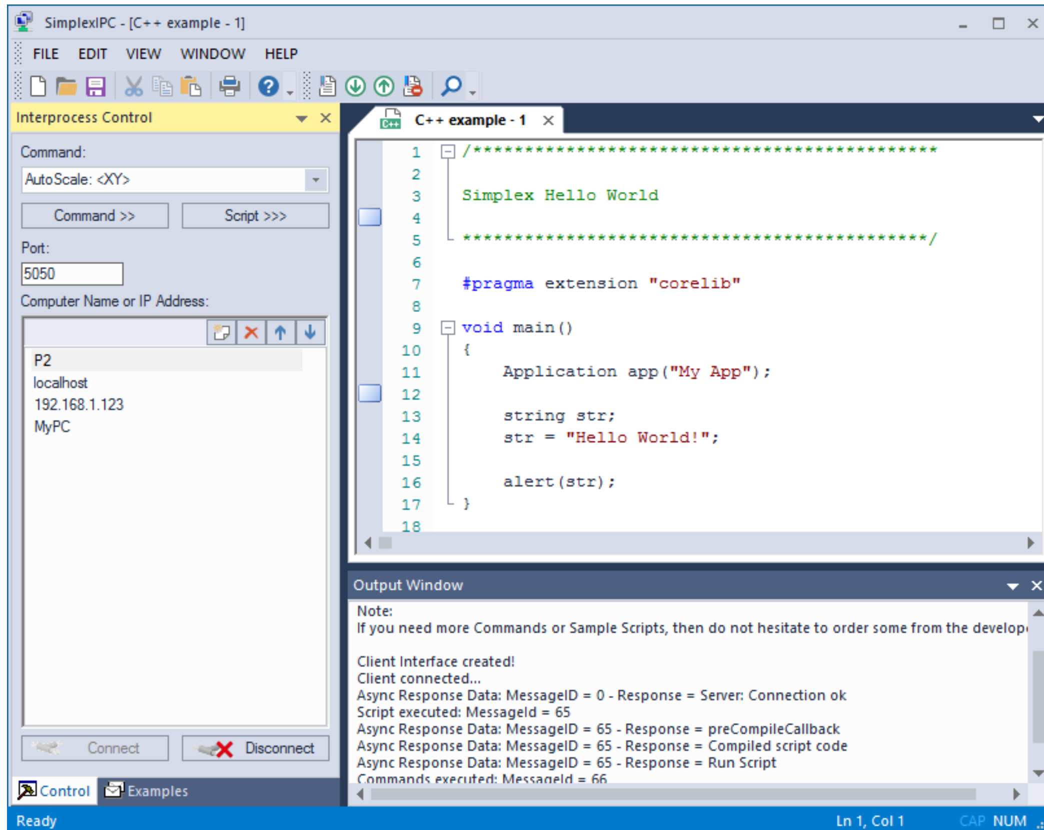
```
C:\Users\Ralf.MyPC\Documents\SimplexNumerica
```

This folder looks like:

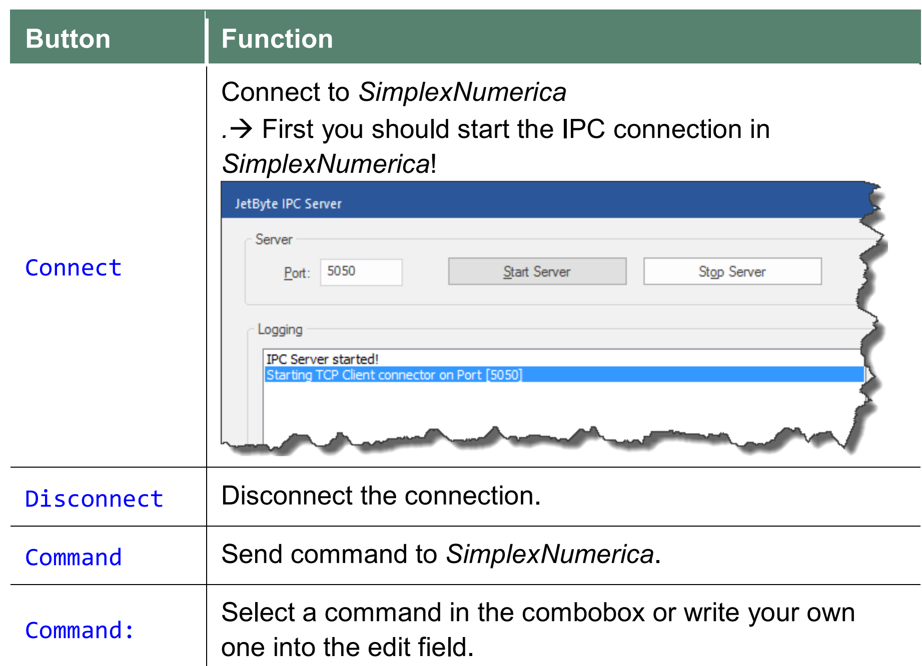


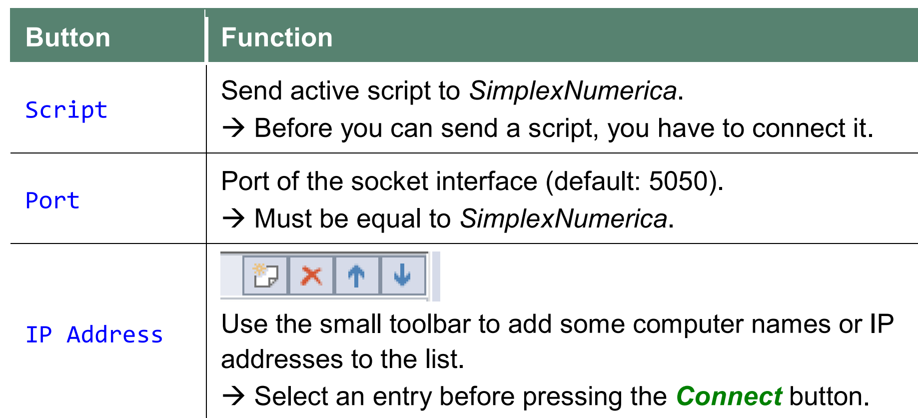
4.1 User Interface

SimplexIPC performs like a text editor. It has a similar look & feel. If you work with *Microsoft Visual Studio*, then you immediately feel comfortable with it.



The left dockable window, called ***Interprocess Control***, takes control over IPC.

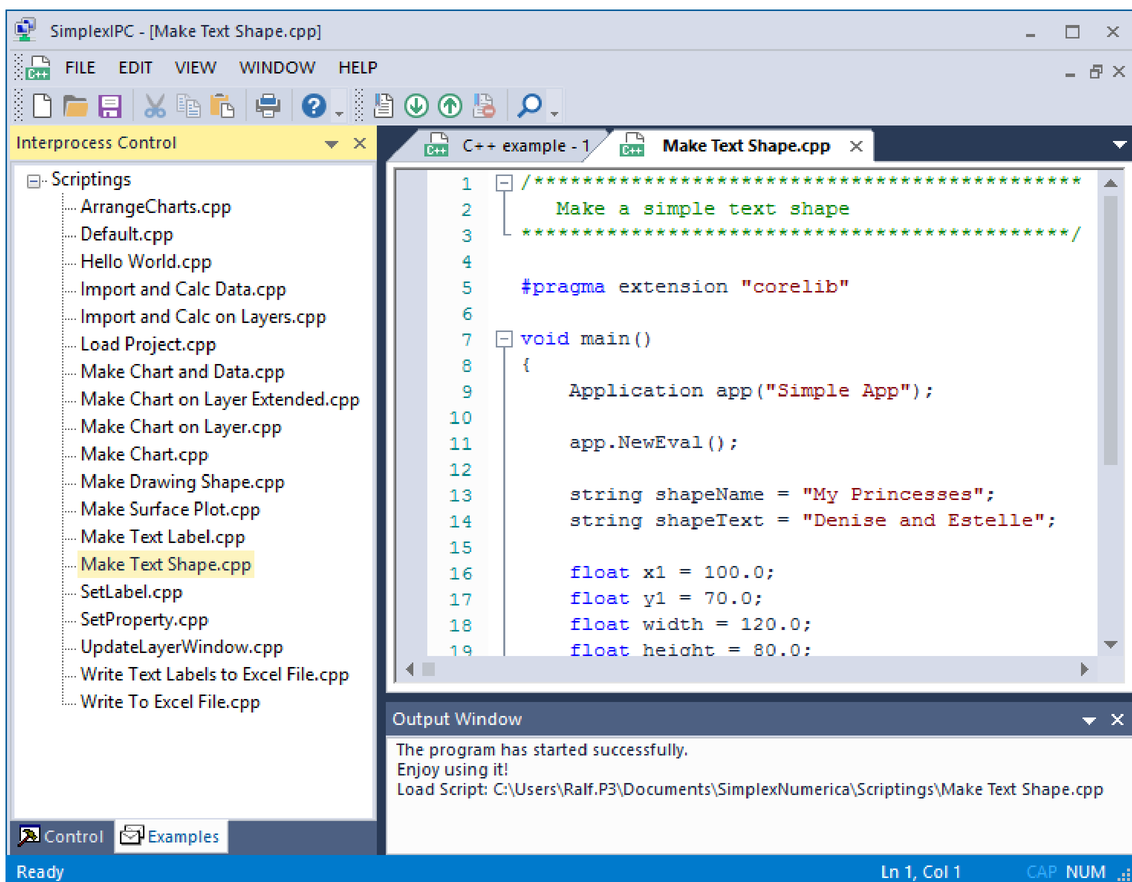
Button	Function
`Connect`	Connect to *SimplexNumerica* .→ First you should start the IPC connection in *SimplexNumerica*! 
`Disconnect`	Disconnect the connection.
`Command`	Send command to *SimplexNumerica*.
`Command:`	Select a command in the combobox or write your own one into the edit field.



Button	Function
`Script`	Send active script to *SimplexNumerica*. → Before you can send a script, you have to connect it.
`Port`	Port of the socket interface (default: 5050). → Must be equal to *SimplexNumerica*.
`IP Address`	Use the small toolbar to add some computer names or IP addresses to the list. → Select an entry before pressing the ***Connect*** button.

```
Info
In this version, only one connection to the same time is permitted!
→ Look to the Output Window for further information.
```

Click on the Tab Examples...

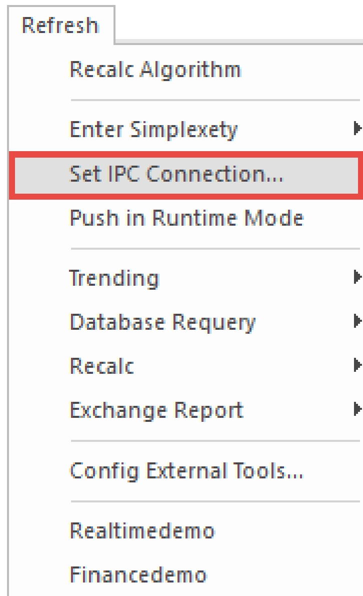


All *.cpp script files in the above mentioned folder

`C:\Users\Ralf.MyPC\Documents\SimplexNumerica\Scriptings`

will be listed here. Click on an entry will open the right MDI window and the code is shown.

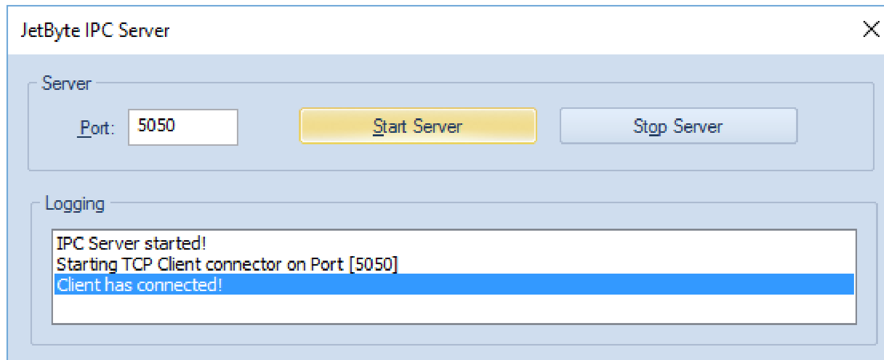
4.2 Send an Example



→ Start *SimplexNumerica* and call the Pulldownmenu ***Set IPC Connection…***

Use this menu item as the starting point for the ***SimplexNumerica Inter Process Control (IPC)*** Client/Server functionality.

→ Push the button ***Start Server***.



That's it in *SimplexNumerica*.

→ Open *SimplexIPC* and setup the right connection. Push the button ***Connect*** and send a Script.

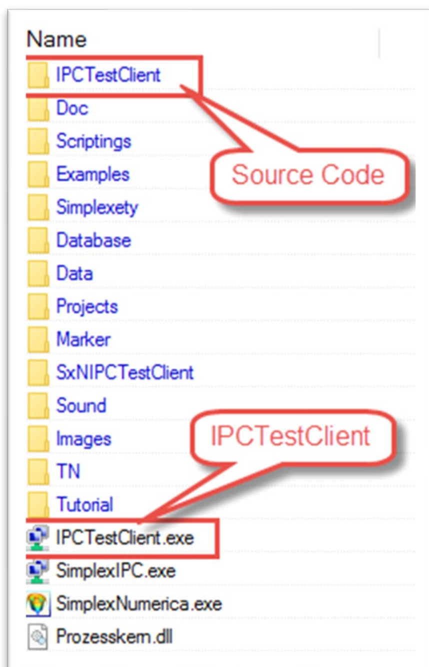
That's it in *SimplexIPC*.

5 IPC Test Client

This is an external test application (incl. source code) to control *SimplexNumerica* remotely from outside. The app is called ***IPCTestClient*** whereby, again, ***IPC*** means *I**nter** **P**rocess **C**ontrol*.

This client is similar to *SimplexIPC* (see previous chapter) but much simpler to understand, because it behaves like a "Hello World" application.

```
Tip
The source code is inclusive and free from copyrights! Feel free to use
it and spread it around the world.
→ Use this source code in your own applications as a sample!
```



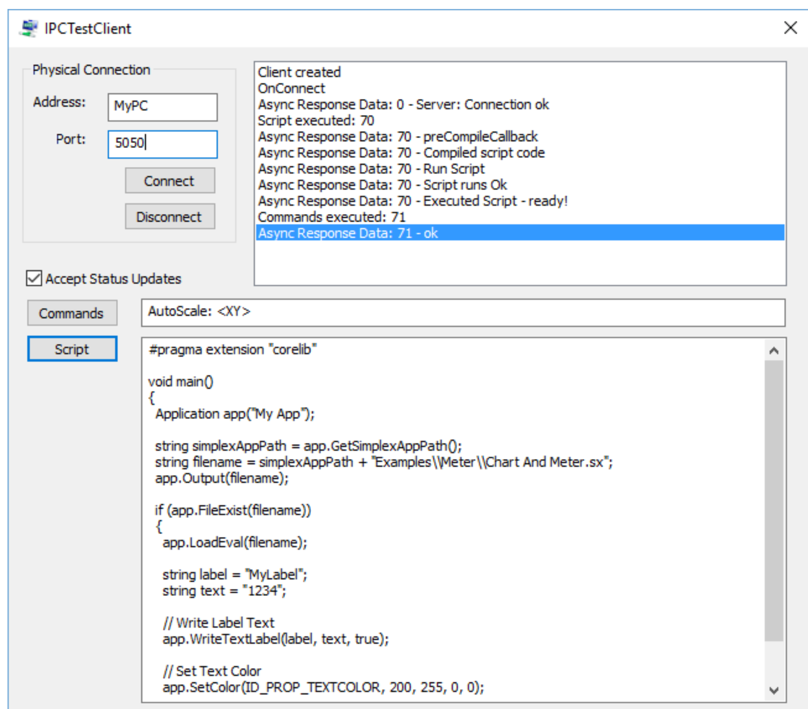
As you can see on the left picture, the *.exe is lying on the install folder, too.

The solution folder with the project files and source code is on the root folder, too.

The C++ solution and project files are for the latest Microsoft Visual Studio 2015.

The project files are available for x64 (64-bit) and i86 (32-bit) both for Unicode Debug and Unicode Release.

Now, start the application…

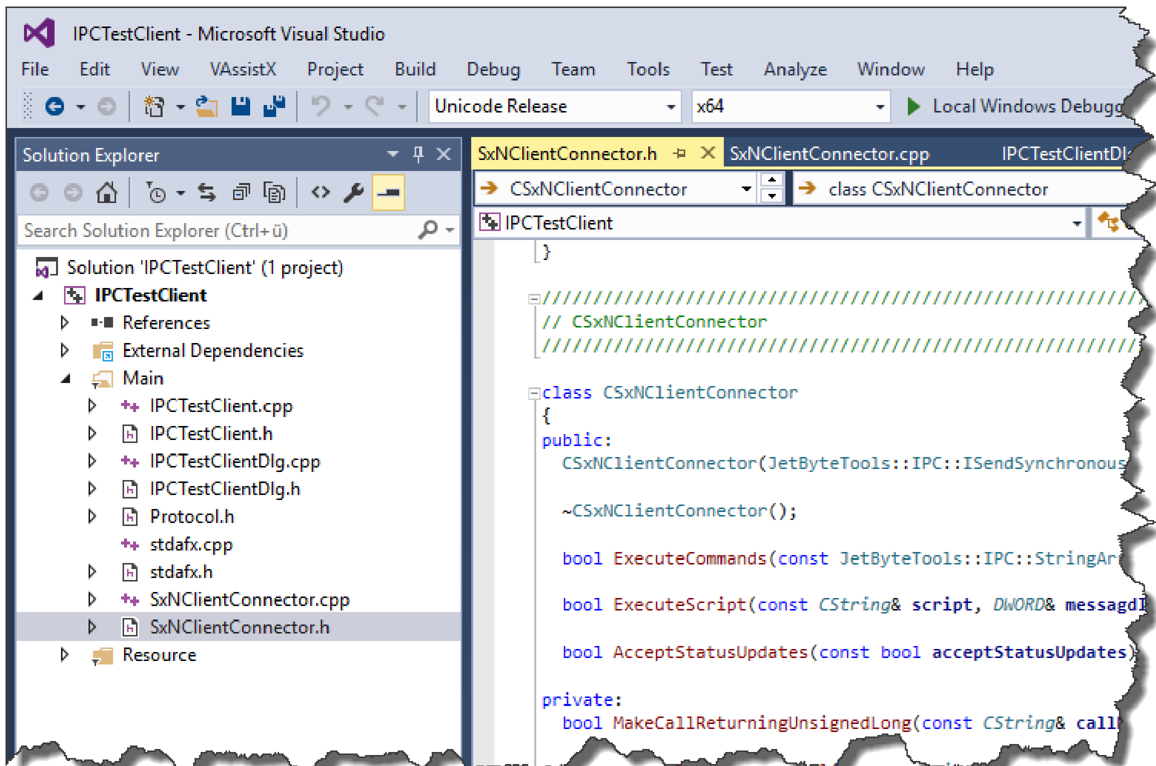


The app is dialog based. The controls are related to *SimplexIPC* (see previous chapter).

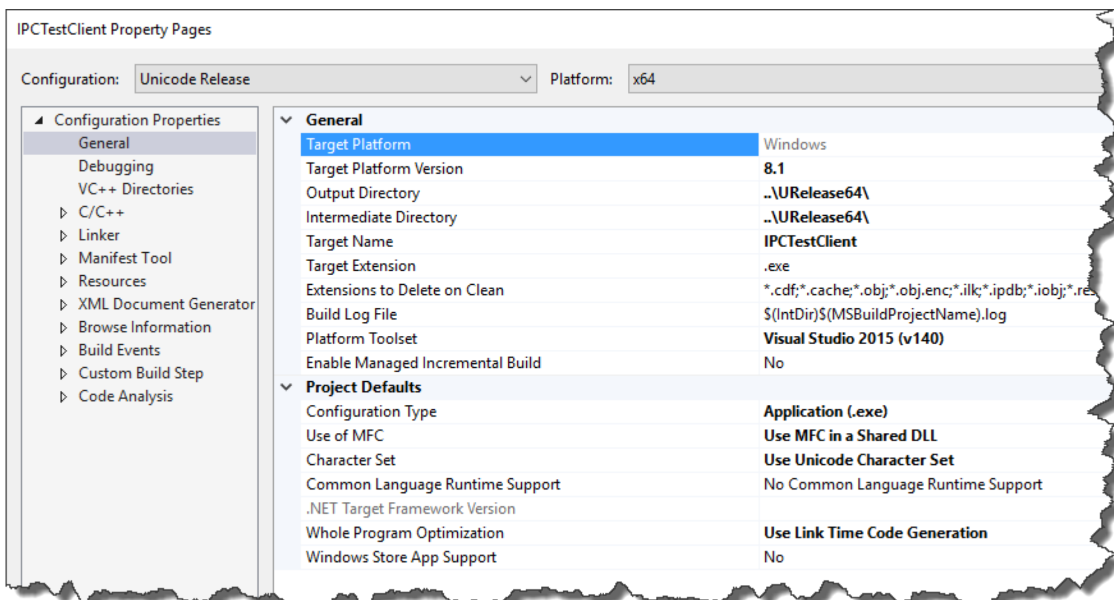
→ Send an example like in previous chapter.

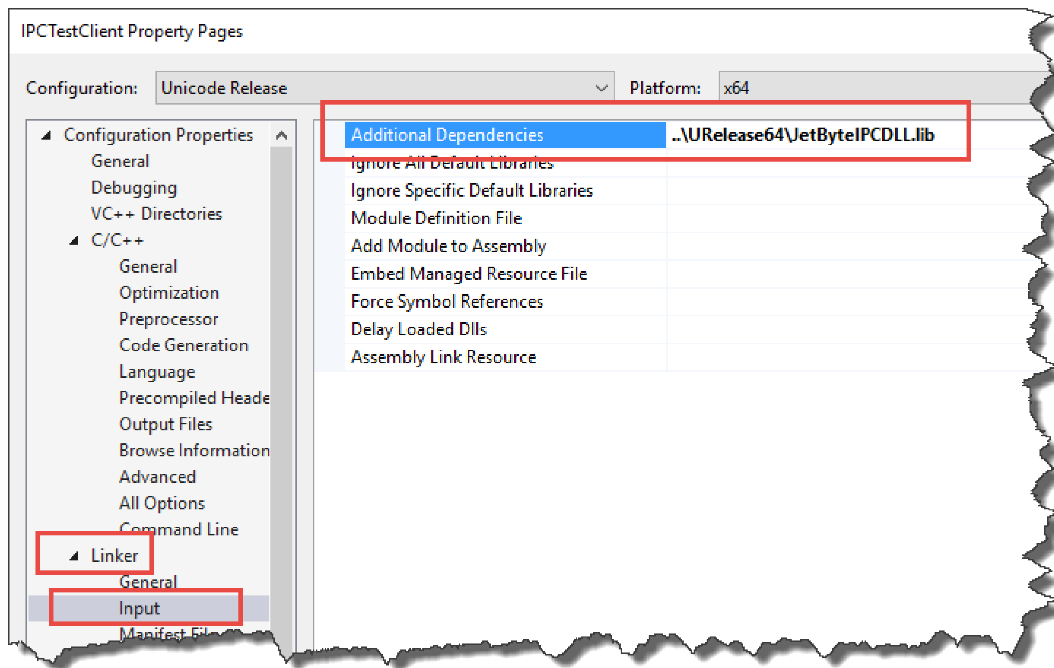
5.1 Source Code

Please open the solution `IPCTestClient.sln` in Microsoft Visual Studio 2015, if available, else make a new solution, dialog based with the help of the wizard and copy the cpp/h/rc files from here into your new folder and see below the screendumps of the properties…



Properties for Microsoft Visual Studio nnnn,   nnnn = 2008, 2010, 2012, (2015) etc.






```
Tip
Do not forget to copy JetByteIPCDLL.lib and JetByteIPCDLL.lib in your
project folder!
```

Try to recompile the solution. If you have trouble, then drop me a mail...

5.2 Usage

Bring this code into your own app and control *SimplexNumerica* remotely via ***Commands & Scripts***.

6 AngelScript

This is a short overview documentation for the ***AngelScript*** scripting language inside *SimplexNumerica* partly grabbed from the original documentation.

As already told at the beginning, ***AngelScript*** is a scripting language with a syntax that's very similar to C++. It is a strictly typed language with many of the types being the same as in C++. This part of the documentation will explain some concepts of how to use ***AngelScript*** in general, but a basic knowledge of the language will be needed to understand all of the concepts.

Please have a look to the *AngelScript* web page at: www.AngelCode.com/AngelScript/ →Maybe there are already updates or changings to the documentation.

Please navigate to the content:

- Globals
- Statements
- Expressions
- Data Types
- Object Handles
- Script Classes
- Operator Precedence
- Reserved Keywords and Tokens
- Strings

I. Globals

- Functions
- Variables
- Classes
- Interfaces
- Imports
- Enums
- Typedefs

II. Statements

- Variable declarations
- Expression statement
- Conditions: if / if-else / switch-case
- Loops: while / do-while / for
- Loop control: break / continue
- Return statement
- Statement blocks

III. Expressions

- Assignments
- Compound assignments
- Function call
- Type conversions
- Math operators
- Bitwise operators
- Logic operators
- Equality comparison operators
- Relational comparison operators
- Identity comparison operators
- Increment operators
- Indexing operator
- Conditional expression
- Member access
- Handle-of
- Parenthesis
- Scope resolution

IV. Data Types

- void
- bool
- Integer numbers
- Real numbers
- Arrays
- Objects
- Object handles
- Strings

V. Object Handles

- Object Handles

VI. Script Classes

- Script classes
- Operator overloads
- Property accessors

VII. Operator Precedence

In expressions, the operator with the highest precedence is always computed first.

6.1 Unary operators

Unary operators have the higher precedence than other operators, and between unary operators the operator closest to the actual value has the highest precedence. Post-operators have higher precedence than pre-operators.

This list shows the available unary operators.

<table>
<tr><td><code>::</code></td><td>scope resolution operator</td></tr>
<tr><td><code>[]</code></td><td>indexing operator</td></tr>
<tr><td><code>++ --</code></td><td>post increment and decrement</td></tr>
<tr><td><code>.</code></td><td>member access</td></tr>
<tr><td><code>++ --</code></td><td>pre increment and decrement</td></tr>
<tr><td><code>not !</code></td><td>logical not</td></tr>
<tr><td><code>+ -</code></td><td>unary positive and negative</td></tr>
<tr><td><code>~</code></td><td>bitwise complement</td></tr>
<tr><td><code>@</code></td><td>handle of</td></tr>
</table>

6.2 Binary and ternary operators

This list shows the dual and ternary operator precedence in decreasing order.

<table>
<tr><td><code>* / %</code></td><td>multiply, divide, and modulo</td></tr>
<tr><td><code>+ -</code></td><td>add and subtract</td></tr>
<tr><td><code><< >> >>></code></td><td>left shift, right shift, and arithmetic right shift</td></tr>
<tr><td><code>&</code></td><td>bitwise and</td></tr>
<tr><td><code>^</code></td><td>bitwise xor</td></tr>
<tr><td><code>|</code></td><td>bitwise or</td></tr>
<tr><td><code><= < >= ></code></td><td>comparison</td></tr>
<tr><td><code>== != is !is xor ^^</code></td><td>equality, identity, and logical exclusive or</td></tr>
<tr><td><code>and &&</code></td><td>logical and</td></tr>
<tr><td><code>or ||</code></td><td>logical or</td></tr>
<tr><td><code>?:</code></td><td>condition</td></tr>
<tr><td><code>= += -= *= /= = &=</code>
<code>|= ^= <<= >>= >>>=</code></td><td>assignment and compound assignments</td></tr>
</table>

IX. Reserved Keywords and Tokens

These are the keywords that are reserved by the language, i.e. they can't be used by any script defined identifiers. Remember that the host application may reserve additional keywords that are specific to that application.

```
and         double      inout       or          uint16
bool        else        int         out         uint32
break       enum        interface   return      uint64
case        false       int8        super*      void
cast        float       int16       switch      while
class       for         int32       this*       xor
const       from*       int64       true
continue    if          is          typedef
default     import      not         uint
do          in          null        uint8
```

* Not really a reserved keyword, but is recognized by the compiler as a built-in keyword.

These are the non-alphabetical tokens that are also used in the language syntax.

```
*           )           =           ~           .
/           ==          ++          <<          &&
%           !=          --          >>          ||
+           ?           &           >>>         !
-           :           ,           &=          [
<=          =           {           |=          ]
<           +=          }           ^=          ^^
>=          -=          ;           <<=         @
>           =           |           >>=         !is
(           /=          ^           >>>=        ::
```

Other than the above tokens there are also numerical, string, identifier, and comment tokens.

```
123456789
123.123e123
123.123e123f
0x1234FEDC
'abc'
"abc"
"""heredoc"""
_Abc123
//
/*
*/
```

The characters space (32), tab (9), carriage return (13), line feed (10), and the UTF8 byte-order-mark (U+FEFF) are all recognized as whitespace.

6.3 Expressions

- Assignments
- Compound assignments
- Function call
- Type conversions
- Math operators
- Bitwise operators
- Logic operators
- Equality comparison operators
- Relational comparison operators
- Identity comparison operators
- Increment operators
- Indexing operator
- Conditional expression
- Member access
- Handle-of
- Parenthesis
- Scope resolution

6.3.1 Assignments

```
lvalue = rvalue;
```

`lvalue` must be an expression that evaluates to a memory location where the expression value can be stored, e.g. a variable. An assignment evaluates to the same value and type of the data stored. The right hand expression is always computed before the left.

6.3.2 Compound assignments

```
lvalue += rvalue;
lvalue = lvalue + rvalue;
```

A compound assignment is a combination of an operator followed by the assignment. The two expressions above means practically the same thing. Except that first one is more efficient in that the lvalue is only evaluated once, which can make a difference if the lvalue is complex expression in itself.

Available operators: `+=  -=  *=  /=  =  &=  |=  ^=  <<=  >>=  >>>=`

6.3.3 Function call

```
func();
func(arg);
func(arg1, arg2);
lvalue = func();
```

Functions are called to perform an action, and possibly return a value that can be used in further operations. If a function takes more than one argument, the argument expressions are evaluated in the reverse order, i.e. the last argument is evaluated first.

6.3.4 Type conversions

```
// implicitly convert the clss handle to a intf handle
intf @a = @clss();
// explicitly convert the intf handle to a clss handle
clss @b = cast<clss>(a);
```

Object handles can be converted to other object handles with the cast operator. If the cast is valid, i.e. the true object implements the class or interface being requested, the operator returns a valid handle. If the cast is not valid, the cast returns a null handle.

The above is called a reference cast, and only works for types that support object handles. In this case the handle still refers to the same object, it is just exposed through a different interface.

Types that do not support object handles can be converted with a value cast instead. In this case a new value is constructed, or in case of objects a new instance of the object is created.

```
// implicit value cast
int a = 1.0f;
// explicit value cast
float b = float(a)/2;
```

In most cases an explicit cast is not necessary for primitive types, however, as the compiler is usually able to do an implicit cast to the correct type.

6.3.5 Math operators

```
c = -(a + b);
```

operator	description	left hand	right hand	result
+	unary positive		*NUM*	*NUM*
-	unary negative		*NUM*	*NUM*
+	addition	*NUM*	*NUM*	*NUM*
-	subtraction	*NUM*	*NUM*	*NUM*
*	multiplication	*NUM*	*NUM*	*NUM*
/	division	*NUM*	*NUM*	*NUM*
%	modulos	*NUM*	*NUM*	*NUM*

Plus and minus can be used as unary operators as well. NUM can be exchanged for any numeric type, e.g. `int` or `float`. Both terms of the dual operations will be implicitly converted to have the same type. The result is always the same type as the original terms. One exception is unary negative which is not available for `uint`.

6.3.6 Bitwise operators

```
c = ~(a | b);
```

operator	description	left hand	right hand	result
~	bitwise complement		*NUM*	*NUM*
&	bitwise and	*NUM*	*NUM*	*NUM*
|	bitwise or	*NUM*	*NUM*	*NUM*
^	bitwise xor	*NUM*	*NUM*	*NUM*
<<	left shift	*NUM*	*NUM*	*NUM*
>>	right shift	*NUM*	*NUM*	*NUM*
>>>	arithmetic right shift	*NUM*	*NUM*	*NUM*

All except ~ are dual operators.

6.3.7 Logic operators

```
if( a and b or not c )
{
  // ... do something
}
```

operator	description	left hand	right hand	result
not	logical not		bool	bool
and	logical and	bool	bool	bool
or	logical or	bool	bool	bool
xor	logical exclusive or	bool	bool	bool

Boolean operators only evaluate necessary terms. For example in expression `a and b`, `b` is only evaluated if `a` is `true`.

Each of the logic operators can be written as symbols as well, i.e. `||` for `or`, `&&` for `and`, `^^` for `xor`, and `!` for `not`.

6.3.8 Equality comparison operators

```
if( a == b )
{
  // ... do something
}
```

The operators `==` and `!=` are used to compare two values to determine if they are equal or not equal, respectively. The result of this operation is always a boolean value.

6.3.9 Relational comparison operators

```
if( a > b )
{
  // ... do something
}
```

The operators <, >, <=, and >= are used to compare two values to determine their relationship. The result is always a boolean value.

6.3.10 Identity comparison operators

```
if( a is null )
{
  // ... do something
}
else if( a is b )
{
  // ... do something
}
```

The operators `is` and `!is` are used to compare the identity of two objects, i.e. to determine if the two are the same object or not. These operators are only valid for reference types as they compare the address of two objects. The result is always a boolean value.

6.3.11 Increment operators

```
// The following means a = i; i = i + 1;
a = i++;
// The following means i = i - 1; b = i;
b = --i;
```

These operators can be placed either before or after an lvalue to increment/decrement its value either before or after the value is used in the expression. The value is always incremented or decremented with 1.

6.3.12 Indexing operator

```
arr[i] = 1;
```

This operator is used to access an element contained within the object. Depending on the object type, the expression between the `[]` needs to be of different types.

6.3.13 Conditional expression

```
choose ? a : b;
```

If the value of `choose` is `true` then the expression returns `a` otherwise it will return `b`. Both `a` and `b` must be of the same type.

6.3.14 Member access

```
object.property = 1;
```

```
object.method();
```

`object` must be an expression resulting in a data type that have members. `property` is the name of a member variable that can be read/set directly. `method` is the name of a member method that can be called on the object.

6.3.15 Handle-of

```
// Make handle reference the object instance
@handle = @object;
// Clear the handle and release the object it references
@handle = null;
```

Object handles are references to an object. More than one handle can reference the same object, and only when no more handles reference an object is the object destroyed.

The members of the object that the handle references are accessed the same way through the handle as if accessed directly through the object variable, i.e. with `.` operator.

6.3.16 Parenthesis

```
a = c * (a + b);
if( (a or b) and c )
{
  // ... do something
}
```

Parenthesis are used to group expressions when the operator precedence does not give the desired order of evaluation.

6.3.17 Scope resolution

```
int value;
void function()
{
  int value;        // local variable overloads the global variable
  ::value = value; // use scope resolution operator to refer to the global
variable
}
```

The scope resolution operator `::` can be used to access variables or functions from another scope when the name is overloaded by a local variable or function. Write the scope name on the left (or blank for the global scope) and the name of the variable/function on the right.

6.4 Strings

(see example script ..\Scriptings\Strings.cpp)

There are two types of string constants supported in the AngelScript language, the normal quoted string, and the documentation strings, called heredoc strings.

The normal strings are written between double quotation marks (") or single quotation marks ('). Inside the constant strings some escape sequences can be used to write exact byte values that might not be possible to write in your normal editor.

Sequence	Value	Description
\0	0	null character
\\	92	back-slash
\'	39	single quotation mark (apostrophe)
\"	34	double quotation mark
\n	10	new line feed
\r	13	carriage return
\t	9	tab character

```
Examples:
string str1 = "This is a string with \"escape sequences".";
```

The heredoc strings are designed for inclusion of large portions of text without processing of escape sequences. A heredoc string is surrounded by triple double-quotation marks ("""), and can span multiple lines of code. If the characters following the start of the string until the first linebreak only contains white space, it is automatically removed by the compiler. Likewise, if the characters following the last line break until the end of the string only contains white space this is also removed.

```
string str = """
This is some text without "escape sequences". This is some text.
This is some text. This is some text. This is some text. This is
some text. This is some text. This is some text. This is some
text. This is some text. This is some text. This is some text.
This is some text.
""";
```

If more than one string constants are written in sequence with only whitespace or comments between them the compiler will concatenate them into one constant.

```
string str = "First line.\n"
             "Second line.\n"
             "Third line.\n";
```

6.4.1 String object and functions

The string object supports a number of operators, and has several class methods and supporting global functions to facilitate the manipulation of strings.

Operators

= assignment

The assignment operator copies the content of the right hand string into the left hand string.

Assignment of primitive types is allowed, which will do a default transformation of the primitive to a string.

+, += concatenation

The concatenation operator appends the content of the right hand string to the end of the left hand string.

Concatenation of primitives types is allowed, which will do a default transformation of the primitive to a string.

==, != equality

Compares the content of the two strings.

<, >, <=, >= comparison

Compares the content of the two strings. The comparison is done on the byte values in the strings, which may not correspond to alphabetical comparisons for some languages.

[] index operator

The index operator gives access to a single byte in the string.

6.4.2 Methods

uint length() const

Returns the length of the string.

void resize(uint)

Sets the length of the string.

bool isEmpty() const

Returns true if the string is empty, i.e. the length is zero.

string substr(uint start = 0, int count = -1) const

Returns a string with the content starting at *start* and the number of bytes given by count. The default arguments will return the whole string as the new string.

void insert(uint pos, const string &in other)

Inserts another string *other* at position *pos* in the original string.

void erase(uint pos, int count = -1)

Erases a range of characters from the string, starting at position *pos* and counting *count* characters.

int findFirst(const string &in str, uint start = 0) const

Find the first occurrence of the value *str* in the string, starting at *start*. If no occurrence is found a negative value will be returned.

int findLast(const string &in str, int start = -1) const

Find the last occurrence of the value *str* in the string. If *start* is informed the search will begin at that position, i.e. any potential occurrence after that position will not be searched. If no occurrence is found a negative value will be returned.

int findFirstOf(const string &in chars, int start = 0) const
int findFirstNotOf(const string &in chars, int start = 0) const
int findLastOf(const string &in chars, int start = -1) const
int findLastNotOf(const string &in chars, int start = -1) const

The first variant finds the first character in the string that matches on of the characters in *chars*, starting at *start*. If no occurrence is found a negative value will be returned.

The second variant finds the first character that doesn't match any of those in *chars*. The third and last variant are the same except they start the search from the end of the string.

```
Note
These functions work on the individual bytes in the strings. They do not
attempt to understand encoded characters, e.g. UTF-8 encoded characters
that can take up to 4 bytes.
```

6.4.3 Functions

array<string> split(const string &in delimiter) const

Splits the string in smaller strings where the delimiter is found.

string join(const array<string> &in arr, const string &in delimiter)

Concatenates the strings in the array into a large string, separated by the delimiter.

int64 parseInt(const string &in str, uint base = 10, uint &out byteCount = 0)
uint64 parseUInt(const string &in str, uint base = 10, uint &out byteCount = 0)

Parses the string for an integer value. The *base* can be 10 or 16 to support decimal numbers or hexadecimal numbers. If *byteCount* is provided it will be set to the number of bytes that were considered as part of the integer value.

double parseFloat(const string &in, uint &out byteCount = 0)

Parses the string for a floating point value. If *byteCount* is provided it will be set to the number of bytes that were considered as part of the value.

string formatInt(int64 val, const string &in options = “”, uint width = 0)
string formatUInt(uint64 val, const string &in options = “”, uint width = 0)
string formatFloat(double val, const string &in options = “”, uint width = 0, uint precision = 0)

The format functions take a string that defines how the number should be formatted. The string is a combination of the following characters:

- l = left justify
- 0 = pad with zeroes
- + = always include the sign, even if positive
- space = add a space in case of positive number
- h = hexadecimal integer small letters (not valid for formatFloat)
- H = hexadecimal integer capital letters (not valid for formatFloat)
- e = exponent character with small e (only valid for formatFloat)
- E = exponent character with capital E (only valid for formatFloat)

Examples:

```
// Left justify number in string with 10 characters
string justified = formatInt(number, “l”, 10);

// Create hexadecimal representation with capital letters, right justified
string hex = formatInt(number, “H”, 10);

// Right justified, padded with zeroes and two digits after decimal separator
string num = formatFloat(number, “0”, 8, 2);
```

6.5 Template Arrays

It is possible to declare array variables with the array identifier followed by the type of the elements within angle brackets.

Example:

```
array<int> a, b, c;
array<Foo@> d;
```

`a`, `b`, and `c` are now arrays of integers, and `d` is an array of handles to objects of the Foo type.

When declaring arrays it is possible to define the initial size of the array by passing the length as a parameter to the constructor. The elements can also be individually initialized by specifying an initialization list.
Example:

```
array<int> a;         // A zero-length array of integers
array<int> b(3);      // An array of integers with 3 elements

// An array of integers with 3 elements, all set to 1 by default
array<int> c(3, 1);

// An array of integers with 3 elements with specific values
array<int> d = {5,6,7};
```

Multidimensional arrays are supported as arrays of arrays, for example:

```
array<array<int>> a;                 // An empty array of arrays of integers
array<array<int>> b = {{1,2},{3,4}} // A 2 by 2 array with initialized values

// A 10 by 10 array of integers with uninitialized values
array<array<int>> c(10, array<int>(10));
```

Each element in the array is accessed with the indexing operator. The indices are zero based, i.e. the range of valid indices are from 0 to length - 1.

```
a[0] = some_value;
```

When the array stores **handles** the elements are assigned using the **handle assignment**.

```
// Declare an array with initial length 1
array<Foo@> arr(1);

// Set the first element to point to a new instance of Foo
@arr[0] = Foo();
```

6.5.1 Array object and functions

The array object supports a number of operators and has several class methods to facilitate the manipulation of strings.

The array object is a reference type even if the elements are not, so it's possible to use handles to the array object when passing it around to avoid costly copies.

Operators

- = assignment
- [] index operator
- ==, != equality

Methods

- uint length() const
- void resize(uint)
- void reverse()
- void insertAt(uint index, const T& in)
- void insertLast(const T& in)
- void removeAt(uint index)
- void removeLast()
- void sortAsc()
- void sortAsc(uint startAt, uint count)
- void sortDesc()
- void sortDesc(uint startAt, uint count)
- int find(const T& in)
- int find(uint startAt, const T& in)
- int findByRef(const T& in)
- int findByRef(uint startAt, const T& in)

The T represents the type of the array elements.

Script example:

```
int main()  {
  array<int> arr = {1,2,3}; // 1,2,3
  arr.insertLast(0);        // 1,2,3,0
  arr.insertAt(2,4);        // 1,2,4,3,0
  arr.removeAt(1);          // 1,4,3,0
  arr.sortAsc();            // 0,1,3,4
  int sum = 0;
  for( uint n = 0; n < arr.length(); n++ )
    sum += arr[n];
  return sum;  }
```

6.6 Data Types

Note that the host application may add types specific to that application, refer to the application's manual for more information.

- void
- bool
- Integer numbers
- Real numbers
- Arrays
- Objects
- Object handles
- Strings

6.6.1 void

`void` is not really a data type, more like lack of data type. It can only be used to tell the compiler that a function doesn't return any data.

6.6.2 bool

`bool` is a boolean type with only two possible values: `true` or `false`. The keywords `true` and `false` are constants of type `bool` that can be used as such in expressions.

6.6.3 Integer numbers

type	min value	max value
`int8`	-128	127
`int16`	-32,768	32,767
`int`	-2,147,483,648	2,147,483,647
`int64`	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
`uint8`	0	255
`uint16`	0	65,535
`uint`	0	4,294,967,295
`uint64`	0	18,446,744,073,709,551,615

As the scripting engine has been optimized for 32 bit datatypes, using the smaller variants is only recommended for accessing application specified variables. For local variables it is better to use the 32 bit variant.

`int32` is an alias for `int`, and `uint32` is an alias for `uint`.

6.6.4 Real numbers

type	range of values	smallest positive value	maximum digits
float	+/- 3.402823466e+38	1.175494351e-38	6
double	+/- 1.7976931348623158e+308	2.2250738585072014e-308	15

Rounding errors will occur if more digits than the maximum number of digits are used.

Curiousity: Real numbers may also have the additional values of positive and negative 0 or infinite, and NaN (Not-a-Number). For float NaN is represented by the 32 bit data word 0x7fc00000.

6.6.5 Arrays

➔ Please have a look to the new dynamic template arrays in chapter 6.5.

It is also possible to declare array variables by appending the [] brackets to the type.

When declaring a variable with a type modifier, the type modifier affects the type of all variables in the list. Example:

```
int[] a, b, c;
```

a, b, and c are now arrays of integers.

When declaring arrays it is possible to define the initial size of the array by passing the length as a parameter to the constructor. The elements can also be individually initialized by specifying an initialization list. Example:

```
int[] a;           // A zero-length array of integers
int[] b(3);        // An array of integers with 3 elements
int[] c = {,3,4,}; // An array of integers with 4 elements, where
                   // the second and third elements are initialized
```

Each element in the array is accessed with the indexing operator. The indices are zero based, i.e the range of valid indices are from 0 to length - 1.

```
a[0] = some_value;
```

An array also has two methods. length() allow you to determine how many elements are in the array, and resize() lets you resize the array.

6.6.6 Objects

There are two forms of objects, reference types and value types.

Value types behave much like the primitive types, in that they are allocated on the stack and deallocated when the variable goes out of scope. Only the application can register these types, so you need to check with the application's documentation for more information about the registered types.

Reference types are allocated on the memory heap, and may outlive the initial variable that allocates them if another reference to the instance is kept. All script declared classes are reference types. Interfaces are a special form of reference types, that cannot be instanciated, but can be used to access the objects that implement the interfaces without knowing exactly what type of object it is.

```
obj o;        // An object is instanciated
o = obj();    // A temporary instance is created whose
              // value is assigned to the variable
```

6.6.7 Object handles

Object handles are a special type that can be used to hold references to other objects. When calling methods or accessing properties on a variable that is an object handle you will be accessing the actual object that the handle references, just as if it was an alias. Note that unless initialized with the handle of an object, the handle is `null`.

```
obj o;
obj@ a;            // a is initialized to null
obj@ b = @o;       // b holds a reference to o
b.ModifyMe();      // The method modifies the original object
if( a is null )    // Verify if the object points to an object
{
  @a = @b;         // Make a hold a reference to the same object as b
}
```

Not all types allow a handle to be taken. Neither of the primitive types can have handles, and there may exist some object types that do not allow handles. Which objects allow handles or not, are up to the application that registers them.

Object handle and array type modifiers can be combined to form handles to arrays, or arrays of handles, etc.

See also:

Object handles

6.6.8 Strings

Strings hold an array of bytes. The only limit to how large this array can be is the memory available on the computer.

There are two types of string constants supported in the ***AngelScript*** language, the normal double quoted string, and the documentation strings, called heredoc strings.

The normal strings are written between double quotation marks (") or single quotation marks (')[1]. Inside the constant strings some escape sequences can be used to write exact byte values that might not be possible to write in your normal editor.

sequence	value	description
\0	0	null character
\\	92	back-slash
\'	39	single quotation mark (apostrophe)
\"	34	double quotation mark
\n	10	new line feed
\r	13	carriage return
\t	9	tab character
\xFF	0xFF	FF should be exchanged for the hexadecimal number representing the byte value wanted
\uFFFF	0xFFFF	FFFF should be exchanged for the hexadecimal number representing the unicode code point
\UFFFFFFFF	0xFFFFFFFF	FFFFFFFF should be exchanged for the hexadecimal number representing the unicode code point

```
string str1 = "This is a string with \"escape sequences\".";
```

The heredoc strings are designed for inclusion of large portions of text without processing of escape sequences. A heredoc string is surrounded by triple double-quotation marks ("""), and can span multiple lines of code. If the characters following the start of the string until the first linebreak only contains white space, it is automatically removed by the compiler. Likewise if the characters following the last line break until the end of the string only contains white space this is also remove, including the linebreak.

```
string str = """
This is some text without "escape sequences". This is some text.
This is some text. This is some text. This is some text. This is
some text. This is some text. This is some text. This is some
text. This is some text. This is some text. This is some text.
This is some text.
""";
```

If more than one string constants are written in sequence with only whitespace or comments between them the compiler will concatenate them into one constant.

```
string str = "First line.\n"
             "Second line.\n"
             "Third line.\n";
```

The escape sequences \u and \U will add the specified unicode code point as a UTF8 encoded sequence. Only valid unicode 5.1 code points are accepted, i.e. code points between U+D800 and U+DFFF (reserved for surrogate pairs) or above U+10FFFF are not accepted.

6.7 Statements

- Variable declarations
- Expression statement
- Conditions: if / if-else / switch-case
- Loops: while / do-while / for
- Loop control: break / continue
- Return statement
- Statement blocks

6.7.1 Variable declarations

```
int var = 0, var2 = 10;
object@ handle, handle2;
const float pi = 3.141592f;
```

Variables must be declared before they are used within the statement block, or any sub blocks. When the code exits the statement block where the variable was declared the variable is no longer valid.

A variable can be declared with or without an initial expression. If it is declared with an initial expression it, the expression must have the evaluate to a type compatible with the variable type.

Any number of variables can be declared on the same line separated with commas, where all variables then get the same type.

Variables can be declared as `const`. In these cases the value of the variable cannot be changed after initialization.

Variables of primitive types that are declared without an initial value, will have a random value. Variables of complex types, such as handles and object are initialized with a default value. For handles this is `null`, for objects this is what is defined by the object's default constructor.

6.7.2 Expression statement

```
a = b;  // a variable assignment
func(); // a function call
```

Any expression may be placed alone on a line as a statement. This will normally be used for variable assignments or function calls that don't return any value of importance.

All expression statements must end with a `;`.

6.7.3 Conditions: if / if-else / switch-case

```
if( condition )
{
  // Do something if condition is true
}
if( value < 10 )
{
```

```
  // Do something if value is less than 10
}
else
{
  // Do something else if value is greater than or equal to 10
}
```

If statements are used to decide whether to execute a part of the logic or not depending on a certain condition. The conditional expression must always evaluate to `true` or `false`.

It's possible to chain several `if-else` statements, in which case each condition will be evaluated sequencially until one is found to be `true`.

```
switch( value )
{
case 0:
  // Do something if value equals 0, then leave
  break;
case 2:
case constant_value:
  // This will be executed if value equals 2 or the constant_value
  break;
default:
  // This will be executed if value doesn't equal any of the cases
}
```

If you have an integer (signed or unsigned) expression that have many different outcomes that should lead to different code, a switch case is often the best choice for implementing the condition. It is much faster than a series of ifs, especially if all of the case values are close in numbers.

Each case should be terminated with a break statement unless you want the code to continue with the next case.

The case value can be a constant variable that was initialized with a constant expression. If the constant variable was initialized with an expression that cannot be determined at compile time it cannot be used in the case values.

6.7.4 Loops: while / do-while / for

```
// Loop, where the condition is checked before the logic is executed
int i = 0;
while( i < 10 )
{
  // Do something
  i++;
}
// Loop, where the logic is executed before the condition is checked
int j = 0;
do
{
  // Do something
  j++;
} while( j < 10 );
```

For both `while` and `do-while` the expression that determines if the loop should continue must evaluate to either true or false. If it evaluates to true, the loop continues, otherwise it stops and the code will continue with the next statement immediately following the loop.

```
// More compact loop, where condituion is checked before the logic is executed
for( int n = 0; n < 10; n++ )
{
  // Do something
}
```

The `for` loop is a more compact form of a `while` loop. The first part of the statement (until the first `;`) is executed only once, before the loop starts. Here it is possible to declare a variable that will be visible only within the loop statement. The second part is the condition that must be satisfied for the loop to be executed. A blank expression here will always evaluate to true. The last part is executed after the logic within the loop, e.g. used to increment an iteration variable.

6.7.5 Loop control: break / continue

```
for(;;) // endless loop
{
  // Do something
  // End the loop when condition is true
  if( condition )
    break;
}
```

`break` terminates the smallest enclosing loop statement or switch statement.

```
for(int n = 0; n < 10; n++ )
{
  if( n == 5 )
    continue;
  // Do something for all values from 0 to 9, except for the value 5
}
```

`continue` jumps to the next iteration of the smallest enclosing loop statement.

6.7.6 Return statement

```
float valueOfPI()
{
  return 3.141592f; // return a value
}
```

Any function with a return type other than `void` must be finished with a `return` statement where expression evaluates to the same data type as the function return type. Functions declared as `void` can have `return` statements without any expression to terminate early.

6.7.7 Statement blocks

```
{
  int a;
  float b;
  {
```

```
    float a; // Override the declaration of the outer variable
             // but only within the scope of this block.
    // variables from outer blocks are still visible
    b = a;
  }
  // a now refers to the integer variable again
}
```

A statement block is a collection of statements. Each statement block has its own scope of visibility, so variables declared within a statement block are not visible outside the block.

6.8 Property Assessors

Many times when working with class properties it is necessary to make sure specific logic is followed when accessing them. An example would be to always send a notification when a property is modified, or computing the value of the property from other properties. By implementing property accessor methods for the properties this can be implemented by the class itself, making it easier for the one who accesses the properties.

In ***AngelScript*** property accessors are implemented as ordinary class methods with the prefixes `get_` and `set_`.

```
// The class declaration with property accessors
class MyObj
{
  int get_prop() const
  {
    // The actual value of the property could be stored
    // somewhere else, or even computed at access time
    return realProp;
  }
  void set_prop(int val)
  {
    // Here we can do extra logic, e.g. make sure
    // the value is within the proper range
    if( val > 1000 ) val = 1000;
    if( val < 0 ) val = 0;
    realProp = val;
  }
  // The caller should use the property accessors
  // 'prop' to access this property
  int realProp;
}
// An example for how to access the property through the accessors
void Func()
{
  MyObj obj;
  // Set the property value just like a normal property.
  // The compiler will convert this to a call to set_prop(10000).
  obj.prop = 10000;
  // Get the property value just a like a normal property.
  // The compiler will convert this to a call to get_prop().
  assert( obj.prop == 1000 );
}
```

When implementing the property accessors you must make sure the return type of the get accessor and the parameter type of the set accessor match, otherwise the compiler will not know which is the correct type to use.

You can also leave out either the get or set accessor. If you leave out the set accessor, then the property will be read-only. If you leave out the get accessor, then the property will be write-only.

6.9 Globals

All global declarations share the same namespace so their names may not conflict. This includes extended data types and built-in functions registered by the host application. Also, all declarations are visible to all, e.g. a function to be called does not have to be declared above the function that calls it.

- Functions
- Variables
- Classes
- Interfaces
- Imports
- Enums
- Typedefs

6.9.1 Functions

Global functions are declared normally, just as in C/C++. The function body must be defined, i.e. it is not possible to declare prototypes, nor is it necessary as the compiler can resolve the function names anyway.

For parameters sent by reference, i.e. with the `&` modifier it is necessary to specify in which direction the value is passed, `in`, `out`, or `inout`, e.g. `&out`. If no keyword is used, the compiler assumes the `inout` modifier. For parameters marked with `in`, the value is passed in to the function, and for parameters marked with `out` the value is returned from the function.

Parameters can also be declared as `const` which prohibits the alteration of their value. It is good practice to declare variables that will not be changed as `const`, because it makes for more readable code and the compiler is also able to take advantage of it some times. Especially for `const &in` the compiler is many times able to avoid a copy of the value.

Note that although functions that return types by references can't be declared by scripts you may still see functions like these if the host application defines them. In that case the returned value may also be used as the target in assignments.

```
int MyFunction(int a, int b)
{
  return a + b;
}
```

6.9.2 Variables

Global variables may be declared in the scripts, which will then be shared between all contexts accessing the script module.

The global variables may be initialized by simple expressions that do not require any functions to be called, i.e. the value can be evaluated at compile time.

Variables declared globally like this are accessible from all functions. The value of the variables are initialized at compile time and any changes are maintained between calls. If a global variable holds a memory resource, e.g. a string, its memory is released when the module is discarded or the script engine is reset.

```
int MyValue = 0;
const uint Flag1 = 0x01;
```

Variables of primitive types are initialized before variables of non-primitive types. This allows class constructors to access other global variables already with their correct initial value. The exception is if the other global variable also is of a non-primitive type, in which case there is no guarantee which variable is initialized first, which may lead to null-pointer exceptions being thrown during initialization.

6.9.3 Classes

In ***AngelScript*** the script writer may declare script classes. The syntax is similar to that of C++ or Java.

With classes the script writer can declare new data types that hold groups of variables and methods to manipulate them. The classes also supports inheritance and polymorphism through interfaces.

```
// The class declaration
class MyClass
{
  // The default constructor
  MyClass()
  {
    a = 0;
  }
  // A class method
  void DoSomething()
  {
    a *= 2;
  }
  // A class property
  int a;
}
```

See also:

Script classes

6.9.4 Interfaces

An interface works like a contract, the classes that implements an interface are guaranteed to implement the methods declared in the interface. This allows for the use of polymorphism in that a function can specify that it wants an object handle to an object that implements a certain interface. The function can then call the methods on this interface without having to know the exact type of the object that it is working with.

```
// The interface declaration
interface MyInterface
{
  void DoSomething();
}
```

```
// A class that implements the interface MyInterface
class MyClass : MyInterface
{
  void DoSomething()
  {
    // Do something
  }
}
```

A class can implement multiple interfaces; Simply list all the interfaces separated by a comma.

6.9.5 Imports

Sometimes it may be useful to load script modules dynamically without having to recompile the main script, but still let the modules interact with each other. In that case the script may import functions from another module. This declaration is written using the import keyword, followed by the function signature, and then specifying which module to import from.

This allows the script to be compiled using these imported functions, without them actually being available at compile time. The application can then bind the functions at a later time, and even unbind them again.

If a script is calling an imported function that has not yet been bound the script will be aborted with a script exception.

```
import void MyFunction(int a, int b) from "Another module";
```

6.9.6 Enums

Enums are a convenient way of registering a family of integer constants that may be used throughout the script as named literals instead of numeric constants. Using enums often help improve the readability of the code, as the named literal normally explains what the intention is without the reader having to look up what a numeric value means in the manual.

Even though enums list the valid values, you cannot rely on a variable of the enum type to only contain values from the declared list. Always have a default action in case the variable holds an unexpected value.

The enum values are declared by listing them in an enum statement. Unless a specific value is given for an enum constant it will take the value of the previous constant + 1. The first constant will receive the value 0, unless otherwise specified.

```
enum MyEnum
{
  eValue0,
  eValue2 = 2,
  eValue3,
  eValue200 = eValue2 * 100
}
```

6.9.7 Typedefs

Typedefs are used to define aliases for other types.

Currently a typedef can only be used to define an alias for primitive types, but a future version will have more complete support for all kinds of types.

```
typedef float  real32;
typedef double real64;
```

6.9.8 Object Handles

An object handle is a type that can hold a reference to an object. With object handles it is possible to declare more than one variables that refer to the same physical object.

Not all types allow object handles to be used. None of the primitive data types, bool, int, float, etc, can have object handles. Object types registered by the application may or may not allow object handles, depending on how they have been registered.

General usage

An object handle is declared by appending the @ symbol to the data type.

```
object@ obj_h;
```

This code declares the object handle obj and initializes it to null, i.e. it doesn't hold a reference to any object.

In expressions variables declared as object handles are used the exact same way as normal objects. But you should be aware that object handles are not guaranteed to actually reference an object, and if you try to access the contents of an object in a handle that is null an exception will be raised.

```
object obj;
object@ obj_h;
obj.Method();
obj_h.Method();
```

Operators like = or any other operator registered for the object type work on the actual object that the handle references. These will also throw an exception if the handle is empty.

```
object obj;
object@ obj_h;
obj_h = obj;
```

When you need to make an operation on the actual handle, you should prepend the expression with the @ symbol. Setting the object handle to point to an object is for example done like this:

```
object obj;
object@ obj_h;
@obj_h = @obj;
```

An object handle can be compared against another object handle (of the same type) to verify if they are pointing to the same object or not. It can also be compared against null, which is a special keyword that represents an empty handle. This is done using the identity operator, `is`.

```
object@ obj_a, obj_b;
if( obj_a is obj_b ) {}
if( obj_a !is null ) {}
```

6.9.9 Object life times

An object's life time is normally for the duration of the scope the variable was declared in. But if a handle outside the scope is set to reference the object, the object will live on until all object handles are released.

```
object@ obj_h;
{
  object obj;
  @obj_h = @obj;
  // The object would normally die when the block ends,
  // but the handle is still holding a reference to it
}
// The object still lives on in obj_h ...
obj_h.Method();
// ... until the reference is explicitly released
// or the object handle goes out of scope
@obj_h = null;
```

Object relations and polymorphing

Object handles can be used to write common code for related types, by means of inheritance or interfaces. This allows a handle to an interface to store references to all object types that implement that interface, similarly a handle to a base class can store references to all object types that derive from that class.

```
interface I {}
class A : I {}
class B : I {}
// Store reference in handle to interface
I @i1 = A();
I @i2 = B();
void function(I @i)
{
  // Functions implemented by the interface can be
  // called directly on the interface handle. But if
  // special treatment is need for a specific type, a
  // cast can be used to get a handle to the true type.
  A @a = cast<A>(i);
  if( a !is null )
  {
    // Access A's members directly
    ...
  }
  else
  {
    // The object referenced by i is not of type A
    ...
  }
}
```

6.10 Script Classes

In ***AngelScript*** the script writer may declare script classes. The syntax is similar to that of C++, except the public, protected, and private keywords are not available. All the class methods must be declared with their implementation, like in Java.

The default constructor and destructor are not needed, unless specific logic is wanted. ***AngelScript*** will take care of the proper initialization of members upon construction, and releasing members upon destruction, even if not manually implemented.

With classes the script writer can declare new data types that hold groups of variables and methods to manipulate them. The class' properties can be accessed directly or through property accessors. It is also possible to overload operators for the classes.

```
// The class declaration
class MyClass
{
  // The default constructor
  MyClass()
  {
    a = 0;
  }
  // Destructor
  ~MyClass()
  {
  }
  // Another constructor
  MyClass(int a)
  {
    this.a = a;
  }
  // A class method
  void DoSomething()
  {
    a *= 2;
  }
  // A class property
  int a;
}
```

AngelScript supports single inheritance, where a derived class inherits the properties and methods of its base class. Multiple inheritance is not supported, but polymorphism is supprted by implementing interfaces.

All the class methods are virtual, so it is not necessary to specify this manually. When a derived class overrides an implementation, it can extend the original implementation by specifically calling the base class' method using the scope resolution operator. When implementing the constructor for a derived class the constructor for the base class is called using the `super` keyword. If none of the base class' constructors is manually called, the compiler will automatically insert a call to the default constructor in the beginning. The base class' destructor will always be called after the derived class' destructor, so there is no need to manually do this.

```
// A derived class
class MyDerived : MyClass
{
```

```
    // The default constructor
    MyDerived()
    {
      // Calling the non-default constructor of the base class
      super(10);
      b = 0;
    }
    // Overloading a virtual method
    void DoSomething()
    {
      // Call the base class' implementation
      MyClass::DoSomething();
      // Do something more
      b = a;
    }
    int b;
  }
```

Note, that since ***AngelScript*** uses automatic memory management, it can be difficult to know exactly when the destructor is called, so you shouldn't rely on the destructor being called at a specific moment. ***AngelScript*** will also call the destructor only once, even if the object is resurrected by adding a reference to it while executing the destructor.

6.11 Operator overloads

It is possible to define what should be done when an operator is used with a script class. While not necessary in most scripts it can be useful to improve readability of the code.

This is called operator overloading, and is done by implementing specific class methods. The compiler will recognize and use these methods when it compiles expressions involving the overloaded operators and the script class.

Unary operators

op	**opfunc**
-	opNeg
~	opCom

When the expression `op a` is compiled, the compiler will rewrite it as `a.opfunc` and compile that instead.

Comparison operators

op	**opfunc**
==	opEquals
!=	opEquals
<	opCmp
<=	opCmp
>	opCmp
>=	opCmp

The `a == b` expression will be rewritten as `a.opEquals(b)` and `b.opEquals(a)` and then the best match will be used. `!=` is treated similarly, except that the result is negated. The opEquals method must be implemented to return a `bool` in order to be considered by the compiler.

The comparison operators are rewritten as `a.opCmp(b) op 0` and `0 op b.opCmp(a)` and then the best match is used. The opCmp method must be implemented to return a `int` in order to be considered by the compiler.

If an equality check is made and the opEquals method is not available the compiler looks for the opCmp method instead. So if the opCmp method is available it is really not necesary to implement the opEquals method, except for optimization reasons.

Assignment operators

op **opfunc**

=	opAssign
+=	opAddAssign
-=	opSubAssign
*=	opMulAssign
/=	opDivAssign
%=	opModAssign
&=	opAndAssign
|=	opOrAssign
^=	opXorAssign
<<=	opShlAssign
>>=	opShrAssign
>>>=	opUShrAssign

The assignment expressions `a op b` are rewritten as `a.opfunc(b)` and then the best matching method is used. An assignment operator can for example be implemented like this:

```
obj@ opAssign(const obj &in other)
{
  // Do the proper assignment
  ...
  // Return a handle to self, so that multiple assignments can be chained
  return this;
}
```

All script classes have a default assignment operator that does a bitwise copy of the content of the class, so if that is all you want to do, then there is no need to implement this method.

Binary operators

op	**opfunc**	**opfunc_r**
+	opAdd	opAdd_r
-	opSub	opSub_r
*	opMul	opMul_r
/	opDiv	opDiv_r
%	opMod	opMod_r
&	opAnd	opAnd_r
|	opOr	opOr_r

^	opXor	opXor_r
<<	opShl	opShl_r
>>	opShr	opShr_r
>>>	opUShr	opUShr_r

The expressions with binary operators `a op b` will be rewritten as `a.opfunc(b)` and `b.opfunc_r(a)` and then the best match will be used.

7 End-user License Agreement

Developer:

- Dipl-Phys.-Ing. Ralf Wirtz
  Mürlenbach/Germany
  Email: support@*SimplexNumerica*.com
  Web: www.*SimplexNumerica*.com

All programs developed by Ralf Wirtz, like SIMPLEXNUMERICA, *SIMPLEXIPC*, SIMPLEXEDITOR AND Simplexety ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL THE AUTHOR BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING INCIDENTAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF THE USE OF THE PROGRAM, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This *SimplexNumerica* software or any *SimplexNumerica* software that is made available to download from a server is the copyrighted work of Ralf Wirtz and/or its suppliers. Use of the Software is governed by the terms of the end user license agreement, if any, which accompanies or is included with the Software ("License Agreement") . An end user will be unable to install any Software that is accompanied by or includes a License Agreement, unless he or she first agrees to the License Agreement terms.

The Software is made available for downloading solely for use by end users according to the License Agreement. Any reproduction or redistribution of the Software not in accordance with the License Agreement is expressly prohibited by law, and may result in severe civil and criminal penalties. Violators will be prosecuted to the maximum extent possible.

WITHOUT LIMITING THE FOREGOING, COPYING OR REPRODUCTION OF THE SOFTWARE TO ANY OTHER SERVER OR LOCATION FOR FURTHER REPRODUCTION OR REDISTRIBUTION IS EXPRESSLY PROHIBITED.

THE SOFTWARE IS WARRANTED, IF AT ALL, ONLY ACCORDING TO THE TERMS OF THE LICENSE AGREEMENT. EXCEPT AS WARRANTED IN THE LICENSE AGREEMENT, RALF WIRTZ HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT.

RESTRICTED RIGHTS LEGEND. Any Software which is downloaded from this Server for or on behalf of the United States of America, its agencies and/or instrumentalities ("U.S. Government"), is provided with Restricted Rights. Use , duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Ralf Wirtz, Kasterstr. 30, 52428 Jülich.

NOTICE SPECIFIC TO DOCUMENTS AVAILABLE ON THIS WEBSITE

Permission to use Documents (such as white papers, press releases, data sheets and FAQs) from this server ("Server") is granted, provided that (1) the below copyright notice appears in all copies and that both the copyright notice and this permission notice appear, (2) use of such Documents from this Server is for

informational and non-commercial or personal use only and will not be copied or posted on any network computer or broadcast in any media, and (3) no modifications of any Documents are made. Use for any other purpose is expressly prohibited by law, and may result in severe civil and criminal penalties. Violators will be prosecuted to the maximum extent possible.

Documents specified above do not include the design or layout of the Ralf Wirtz website or any other Ralf Wirtz owned, operated, licensed or controlled site. Elements of Ralf Wirtz Web sites are protected by trade dress and other laws and may not be copied or imitated in whole or in part. No logo, graphic , sound or image from any Ralf Wirtz website may be copied or retransmitted unless expressly permitted by Ralf Wirtz.

RALF WIRTZ AND/OR ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THE INFORMATION CONTAINED IN THE DOCUMENTS AND RELATED GRAPHICS PUBLISHED ON THIS SERVER FOR ANY PURPOSE. ALL SUCH DOCUMENTS AND RELATED GRAPHICS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. RALF WIRTZ AND/OR ITS RESPECTIVE SUPPLIERS HEREBY DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS INFORMATION, INCLUDING ALL IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL RALF WIRTZ AND/OR ITS RESPECTIVE SUPPLIERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF INFORMATION AVAILABLE FROM THIS SERVER.

THE DOCUMENTS AND RELATED GRAPHICS PUBLISHED ON THIS SERVER COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN. RALF WIRTZ AND/OR ITS RESPECTIVE SUPPLIERS MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED HEREIN AT ANY TIME.

NOTICES REGARDING SOFTWARE, DOCUMENTS AND SERVICES AVAILABLE ON THIS WEBSITE.

IN NO EVENT SHALL RALF WIRTZ AND/OR ITS RESPECTIVE SUPPLIERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTUOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF SOFTWARE, DOCUMENTS, PROVISION OF OR FAILURE TO PROVIDE SERVICES, OR INFORMATION AVAILABLE FROM THIS SERVER.

LINKS TO THIRD PARTY SITES

THE LINKS IN THIS AREA WILL LET YOU LEAVE RALF WIRTZ'S SITE. THE LINKED SITES ARE NOT UNDER THE CONTROL OF RALF WIRTZ AND RALF WIRTZ IS NOT RESPONSIBLE FOR THE CONTENTS OF ANY LINKED SITE OR ANY LINK CONTAINED IN A LINKED SITE. RALF WIRTZ IS PROVIDING THESE LINKS TO YOU ONLY AS A CONVENIENCE, AND THE INCLUSION OF ANY LINK DOES NOT IMPLY ENDORSEMENT BY RALF WIRTZ OF THE SITE.

COPYRIGHT NOTICE.
Copyright © 1988-2021 Dipl.-Phys.-Ing. Ralf Wirtz, Hinter Herschenhaus, Mürlenbach/Eifel.
All rights reserved.
TRADEMARKS. Microsoft, Windows, Windows NT, MSN, The Microsoft Network and/or other Microsoft

products referenced herein are either trademarks or registered trademarks of Microsoft. Other product and company names mentioned herein may be the trademarks
of their respective owners.

The names of companies, products, people, characters and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product or event, unless otherwise noted.

Any rights not expressly granted herein are reserved.