Programming in Origin

Origin C

OriginLab Corporation
Copyright © 2002 by OriginLab Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of OriginLab Corporation.

OriginLab, Origin, and LabTalk are either registered trademarks or trademarks of OriginLab Corporation. Other product and company names mentioned herein may be the trademarks of their respective owners.

OriginLab Corporation
One Roundhouse Plaza
Northampton, MA 01060
USA
(413) 586-2013
(800) 969-7720
Fax (413) 585-0126
www.OriginLab.com
# Contents

## Introduction

### Origin C Language Reference

- ANSI C Support
- C++ Support
- C# Support
- Supported Basic and Derived Types
- Built-in Origin C Classes

### Origin C Class Reference

- CategoricalData
- CategoricalMap
- Collection
- Column
- complex
- Curve
- DataObject
- DataPlot
- Dataset
- Datasheet
- file
- Folder
- GraphLayer
- GraphObject
- GraphPage
- Layer
- Layout
- LayoutPage
- matrix - 2D Dynamic Array
- Matrix - Origin Object
- MatrixLayer
- MatrixObject
- MatrixPage
- Note
- OriginObject
- Page
- PageBase
- progressBox
- Project
- ROIObject
- stdioFile
- string
- StyleHolder
- vector
- waitCursor
- Worksheet
- WorksheetPage
Origin C

Introduction

Origin C is an ANSI C compatible programming language that also includes elements of C++ and C#. It provides built-in C++ classes for programmatic access to most Origin objects. It also includes a wide selection of numerical computational routines from the NAG C Library. You can thus program Origin C functions to automate your work, add new functionality, and perform simulations in Origin.

Code Builder is Origin C's integrated development environment. Code Builder provides modern tools to write, compile, test, and debug your code.

Once compiled, your Origin C functions are accessible from Origin. You can call them from the Script window and other Origin dialogs. You can also create interface elements such as buttons or new menu commands to provide easy access to your functions.

Origin C Language Reference

ANSII C Support

Origin C supports a nearly complete ANSI C language syntax, including support of ANSI C elements, program structure (although the main function is not used in Origin C), declarations and types, expressions and assignments, statements, functions, and syntax.

Origin C does not support the following ANSI C features:

1) Origin C only supports one-dimensional arrays. Thus, the following declaration is not supported:
   
   ```
   int aa[5][6];
   ```
   
   Instead, you can declare a matrix object for support of dynamic two-dimensional arrays:
   
   ```
   matrix<int> aa(5, 6);
   ```

2) Function pointers are not supported.

3) Origin C does not support the use of the extern storage class specifier on functions.
4) Origin C does not support macros with multiple arguments, so that following will not work in Origin C:

```c
#define MAX(a, b)  ( (a) > (b) ? (a) : (b) )
```
However, Origin C does support macros with one argument or no argument.

Also, Origin C differs from ANSI C in that Origin C treats ^ as an exponentiate operator. Thus, \( x^y \) is equivalent to `pow(x, y)`.
To treat ^ as the ANSI C bitwise exclusive OR operator in your Origin C function, you can use the `xor` pragma:

```c
#pragma xor(push, FALSE)
```

## C++ Support

In addition to ANSI C support, Origin C supports a subset of C++ features, including the following:

1) Variable declarations are not restricted to the beginning of a block.
   The following example code illustrates this.
   ```c
   void test()
   {
     vector<double> v1;
     v1.SetSize(10);
     for (int ii = 0; ii < 10; ii++)
     {
       v1[ii] = rnd(ii);
     }
     double val = v1[0];
     printf("%f\n", val);
   }
   ```

2) Overloaded function names.
   The following example code defines the `max` function twice, with different formal parameters and return values. The correct function is called based on the arguments in the calling function.
**double max(double, double);**
**int max(int, int);**

**void testOverload()**
{
    int ii = max( 20, 5 );
    double dd = max( 15.5, 30.0 );
    printf("The max int is %d\n", ii);
    printf("The max double is %.1f\n", dd);
}

double max(double d1, double d2)
{
    return ( d1 > d2 ) ? d1 : d2;
}

int max(int i1, int i2)
{
    return ( i1 > i2 ) ? i1 : i2;
}

3) Limited support for classes (internal classes and DLL extended classes).
Origin C provides classes for basic operations and for accessing Origin objects. These classes are prototyped in the header files located in the \OriginC\System subfolder.

4) Passing arguments to functions by reference.
The following example illustrates how you can pass a variable (in this case index) by reference to a function.
void changeVals(int in1, int &in2);

void test()
{
    int val1 = 5, val2 = 10;
    printf("The values are %3d %3d\n", val1, val2);
    changeVals(val1, val2);
    printf("The values are %3d %3d\n", val1, val2);
}

void changeVals(int in1, int &in2)
{
    in1 = in1 + 100;
    in2 = in2 + 100;
    printf("The values are %3d %3d\n", in1, in2);
}

5) Default argument values.
Origin C supports functions with default arguments as illustrated in the following example.
int getVolume(int length, int width = 2, int height = 3);

void test()
{
    int x = 10, y = 12, z = 15;
    out_int("dimensions are: ", getVolume(x, y, z));
    out_int("dimensions are: ", getVolume(x, y));
    out_int("dimensions are: ", getVolume(x));
}

int getVolume(int length, int width, int height)
{
    return length * width * height;
}
C# Support

Origin C also supports a subset of C# features, including the following:

1) Support of collections.
Origin C provides collections for efficient looping through objects of the same type using \texttt{foreach}. For example, the \texttt{Project} class provides collections for all the windows in a project, and for all the worksheet, graph, notes, matrix, and layout windows in a project. Other classes also provide collections. For example, the \texttt{Page} class provides a collection of all the layers on a page.

2) Support of the \texttt{foreach} statement.
Origin C provides a \texttt{foreach} statement to loop through elements in a collection. The following example illustrates the use of the \texttt{foreach} statement.

```c
void setMyCols()
{
    Worksheet wks("data1"); // declare object
    // wks.Columns is collection of columns
    foreach(Column cols in wks.Columns)
    {
        cols.SetDigitMode(DIGITS_SIGNIFICANT);
        cols.SetDigits(10);
    }
}
```

3) Support of the \texttt{using} statement.
You can use the \texttt{using} statement to create a shortcut for calling member functions from a base class. For example, in the following code, the \texttt{using} statement assigns \texttt{wks.GetPage()} to \texttt{wpg}.

```c
void test()
{
    Worksheet wks;
    //wpg is a "shorthand" for wks.GetPage:
    using wpg = wks.GetPage();
    if(wks.Attach("Data1"))
        out_str(wpg.GetName()); //outputs "Data1"
    if(wks.Attach("Data2"))
        out_str(wpg.GetName()); //outputs "Data2"
}
```
Origin C supports the following basic types: integer, floating point, void, character, and string basic types.

**Integer Data Types**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Values</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>-2,147,483,648 to 2,147,483,647</td>
<td>4 bytes</td>
</tr>
<tr>
<td>synonyms include signed, signed int, LONG, long, BOOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned int</td>
<td>0 to 4,294,967,295</td>
<td>4 bytes</td>
</tr>
<tr>
<td>synonyms include unsigned, UINT, DWORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>-32,768 to 32,767</td>
<td>2 bytes</td>
</tr>
<tr>
<td>synonyms include short int, signed short int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsigned short</td>
<td>0 to 65,535</td>
<td>2 bytes</td>
</tr>
<tr>
<td>synonyms include unsigned short int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WORD</td>
<td>16-bit unsigned integer</td>
<td>16-bit</td>
</tr>
<tr>
<td>BYTE</td>
<td>8-bit integer that is not signed.</td>
<td>8-bit</td>
</tr>
</tbody>
</table>

**Floating Point Data Types**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Values</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>1.7E +/- 308 (15 digits)</td>
<td>8 bytes</td>
</tr>
<tr>
<td>float</td>
<td>3.4E +/- 38 (7 digits)</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

**Character(s) Data Types**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Values</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>one ASCII character enclosed in single quotation marks</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

The `void` type is used for functions that do not return a value.
Origin C also supports the ANSI C supported derived types including one-dimensional arrays of variables or objects, pointers to variables or objects of a specific type, references to objects (address of an object), structures, and enumerations.

Exceptions to the standard ANSI C support are the following:

=> Origin C does not support multi-dimensional arrays. However, you can declare a matrix object for support of dynamic two-dimensional arrays:
   matrix<int> aa(5, 6);
=> Origin C does not support function pointers.

Origin C also provides some support for C++ class objects. Origin C provides classes for creating program objects to access and perform operations on Origin objects such as worksheets, columns, data sets, graphs, and layers. Furthermore, OriginPro provides a Class Extender tool to add custom classes into Origin C classes with external DLLs.

**Built-in Origin C Classes**

In addition to most standard ANSI C features, Origin C provides a number of built-in C++ classes. These built-in classes are used to implement several composite data types and to provide programmatic access to many different objects found within the Origin software.

=> Composite data types, not necessarily unique to Origin, implemented by built-in Origin C classes include such things as vectors, matrices, strings, complex numbers, and files.

=> Other objects, more specific to the Origin software, accessed by the built-in Origin C classes include such things as worksheet and graph pages, layers, scales (axes), text/graphic annotations, and notes and layout windows. Origin projects and user interface objects like Project Explorer, wait cursors, and progress dialog boxes can also be accessed by built-in Origin C classes.

Origin C classes that relate to real Origin objects are always listed with the first letter capitalized.

When coding your functions, declarations of objects follow standard C++ syntax. When creating vector, Dataset, Curve, matrix, Matrix, or complex objects, the base type is assumed to be double unless a base type identifier is included in the declaration. Thus, the following two declarations are identical:

```
Dataset<double> s1;
Dataset s1;
```

To declare an object of base type char, int, short, float, BYTE, WORD, UINT, or string (vector class only), include the base type identifier in the declaration as follows:

```
Dataset<int> s1;
vector<string> aa;
```

To simplify your code, vector<string> has been typedefed as stringvector. Thus, the previous example can be written as:

```
stringvector aa;
```
Mapping Real Origin Objects to their Origin C Classes

worksheet page (window) => use WorksheetPage class and inherited functions.

columns => use Column class and inherited functions.

worksheet on page => use Worksheet class and inherited functions.

data in columns => use Dataset class and inherited functions.

matrix page (window) => use MatrixPage class and inherited functions.

properties of matrix => use MatrixObject class and inherited functions.

matrix on page => use MatrixLayer class and inherited functions.

data in matrix => use Matrix class and inherited functions.
graph page (window) => use GraphPage class and inherited functions.

layers on page => use GraphLayer class and inherited functions.

data plots in layer => use DataPlot class and inherited functions for plot details. => use Curve class and inherited functions for XY data pairs.

annotations => use GraphObject class and inherited functions.

data plot style holders saved with templates => use StyleHolder class.

Project Explorer => use Folder class.
Origin project => use Project class.

Class Hierarchy

*Origin Window-related Classes*

- **OriginObject**
  - **PageBase**
  - **Layer**
  - **GraphObject**
    - **DataObject**
      - **Page**
      - **Note**
      - **Layout**
      - **GraphLayer**
      - **Datasheet**
      - **ROIObject**
      - **StyleHolder**
      - **DataPlot**
      - **Column**
      - **MatrixObject**
      - **MatrixPage**
      - **GraphPage**
      - **LayoutPage**
      - **WorksheetPage**
      - **Worksheet**
      - **MatrixLayer**
Data Type Classes

vector  matrix  complex  CategoricalMap  string  file

Dataset  Matrix

CategoricalData  Curve

Utility Classes

Project  Collection  Folder  progressBox  waitCursor

Origin C Class Reference

CategoricalData

Use the CategoricalData class to set a data set as categorical (Column: Set as Categorical).

Inheritance:
vector
Dataset
CategoricalData

Example Code:

The following code sets Data1_c to categorical.

string strData = "data1_c";
CategoricalData cd(strData);

For another code example, see "CategoricalMap".
**CategoricalMap**

Use the **CategoricalMap** class to control how categorical data is used to display data plots. For example, member functions are provided to arrange the categorical data in ascending or descending order (see example).

*Inheritance:*
None.

*Example Code:*

The following sample function sets Data1_A as Categorical and plots the data.

```c
void test(bool bAlphabetical = TRUE)
{
    Dataset y("Data1_B");
    CategoricalData x("Data1_A", CMT_NOMINAL);
    if(!bAlphabetical)
        x.Map.ArrangeZA();
    else
        x.Map.ArrangeAZ();
    GraphLayer gl("Graph1");
    if(x && y && gl)
    {
        Curve cc("Data1_A","Data1_B");
        gl.AddPlot(cc);
    }
}
```

**Collection**

The **Collection** class provides functions for collections, including a function to count the members of a collection and two functions to assign an item in a collection (referenced by index or by name) to an object.

*Inheritance:*
None.
Example Code:

The following code counts the number of layers in Graph1 and outputs the result to the current output window. pp.Layers is the collection of all layers in Graph1, while pp.Layers.Count(); calls the Count function of the collection (all collections have this method).

```c
void getnumlayers()
{
    Page pp("Graph1");
    UINT num = pp.Layers.Count();
    out_int("number of layers = ", num);
}
```

Column

Use the Column class to control the format (Worksheet Column Format dialog box properties) of a column, and to set its width, name, and label.

Inheritance:
OriginObject
    DataObject
        Column

Example Code:

The following code sets the name, display type, display format, and width for the first column in the Data1 worksheet. Note that wks.Columns is the collection of columns in the worksheet. wks.Columns(0) is the first column in the worksheet.

```c
Worksheet wks("Data1");
string strName = "MyTimeData";
wks.Columns(0).SetName(strName);
wks.Columns(0).SetFormat(OKCOLTYPE_TIME);
wks.Columns(0).SetSubFormat(2);
wks.Columns(0).SetWidth(20);
```

The Dataset class is the data object class for a Column. For more information, see "Dataset" on page 16.
**complex**

Origin C supports complex numbers through the `complex` class.

*Inheritance:*
None.

*Example Code:*

The following example outputs a complex value.
```c
complex a = 1+2i;
complex b = sin(a);
out_complex("sin(1+2i)=", b);
```

**Curve**

Use the `Curve` class to access Origin data plots or associations of data sets in the worksheet. A `Curve` is typically a data set with an associated X data set - but not necessarily. You can plot a column against row number and the associated `Curve` will have no X data.

*Inheritance:*
```c
vector
   Dataset
       Curve
```

*Example Code:*

The following example takes a data set name and integer value as arguments, and then performs a polynomial fit and outputs the parameter results.
void myFit(string strData, int nOrder)
{
    Curve cc(strData);
    double paras[10];
    fitpoly(cc, nOrder, paras);
    printf("%d order polynomial fit to %s\n", nOrder, strData);
    for(int ii = 0; ii <= nOrder; ii++)
    {
        printf("P%d\t%4.2f\n", ii, paras[ii]);
    }
}

To call this function from Origin, you can execute the following command, where %C is a system variable that holds the name of the active data set. In this case a second order polynomial fit is performed.
myFit %C 2;

**DataObject**

The **DataObject** class provides member functions to set and get the data range.

*Inheritance:*
OriginObject
    DataObject

*Example Code:*

The following code example sets the range of all the columns in the Data1 worksheet.
void test()
{
    Worksheet wks("Data1");
    foreach(Column cols in wks.Columns)
    {
        BOOL bOK = cols.SetRange(10, 15);
    }
}
**DataPlot**

Use the DataPlot class to set the data plot color.

*Inheritance:*

```
OriginObject
    DataObject
        DataPlot
```

*Example Code:*

The following example creates a new graph window from a custom template which is located in the Origin folder. It then adds a data plot to the first layer, and sets the color to the third color in the color list.

```c
void plotColor()
{
    GraphPage grph;
    string myTemp = "myScatter.otp";
    BOOL bOK = grph.Create(myTemp, CREATE_VISIBLE);
    if (!bOK)
        return;
    GraphLayer grlay = grph.Layers(0);
    Curve cv("Data1_b");
    int nPlot = grlay.AddPlot(cv);
    if(nPlot>=0)
        grlay.DataPlots(nPlot).SetColor(2, TRUE);
}
```

**Dataset**

Use the Dataset class to access Origin data sets.

*Inheritance:*

```
vector
    Dataset
```

*Example Code:*

The following example fills the first and second columns in the Data1 worksheet with values.
void fillColumns()
{
    Dataset s1;
    //specify wks, col num:
    if (s1.Attach("data1", 0))
    {
        s1.SetSize(100);
        for (int ii = 0; ii < 100; ii++)
            s1[ii] = ii;
    }
    //specify data set name:
    if (s1.Attach("data1_b"))
    {
        s1.SetSize(100);
        for (int ii = 0; ii < 100; ii++)
            s1[ii] = ii^2;
    }
}

As this example illustrates, after you declare a Dataset object you can access values of the data set that are attached to that object using the following notation:

objectName[ElementNumber]

where ElementNumber starts at 0 (which contrasts with LabTalk, in which the index starts at 1).

Datasheet

Datasheet is the base class for worksheet and matrix layers in Origin. It includes functions to create new worksheets or matrices, and to get the number of columns and rows.

Inheritance:

OriginObject
    Layer
        DataSheet

Example Code:

The following code outputs the number of columns in the Data1 worksheet to the current output window.

Worksheet wks("data1");
int ncols = wks.GetNumCols();
out_int("ncols = ", ncols);
**file**

Origin C provides the `file` and the `stdioFile` classes for reading and writing to files. The Origin C `file` class has nearly the same functionality as the `CFile` class documented in the Microsoft Foundation Class Library. Similarly, the Origin C `stdioFile` class provides nearly the same functionality as the `CStdioFile` class. The `stdioFile` class provides buffered access to the file and is thus faster than the `file` class, which uses a handle.

*Inheritance:*

None.

*Example Code:*

To review example functions that illustrate how you can import worksheet data to a binary file, see "Importing Data" on page 52.

**Folder**

Use the `Folder` class to access Project Explorer. The `Folder` class includes a `Subfolders` collection of type `Folder`. These are all the `Subfolders` objects in a `Folder` (all the subfolders in a folder). Additionally, the `Folder` class includes a `Pages` collection of type `PageBase`. These are all the `PageBase` objects in a `Folder` (all the windows in a folder). (The `Project` class also provides `RootFolder` and `CurrentFolder` properties.)

*Inheritance:*

None.

*Example Code:*

The following code creates a `Folder` object and assigns it the current Project Explorer folder. It then uses a `foreach` statement to output all folder’s subfolder names to the current output window. It then uses a second `foreach` statement to output the names of all the windows in the current folder to the output window.
void test()
{
    Project prj();
    String strName;
    Folder fld = prj.CurrentFolder;
    foreach(Folder sub in fld.Subfolders)
    {
        strName = sub.GetName();
        printf("<Folder> %s\n", strName);
    }
    foreach(PageBase page in fld.Pages)
    {
        strName = page.GetName();
        printf(" %s\n", strName);
    }
}

The next example creates a Folder object named root and assigns it the root Project Explorer folder. A second Folder object named ff is then created and assigned the Project Explorer folder that contains the Graph1 window. If the Graph1 window is located in the root folder, then that information is output to the current output window. Otherwise, the name of the folder containing Graph1 is output.

void test()
{
    Project prj();
    Folder root = prj.RootFolder;
    Folder ff = root.FindWindow("Graph1");
    if(ff.IsRootFolder())
        printf("Graph1 is in the root folder");
    else
    {
        string path = ff.GetPath();
        printf("Graph1 is in the folder: %s", path);
    }
}

GraphLayer

Use this class to access properties of graph layers. Functions are provided to add data plots to a layer, add error bars or data labels to a data plot, and group data plots in a layer. A function is also provided to access data plots in a graph layer. The GraphLayer class also provides a DataPlots collection
of type DataPlot, which is a collection of all the data plots in the graph layer. It also provides a StyleHolders collection of type StyleHolder, which is a collection of all the styleholders in the graph layer.

Inheritance:

OriginObject
   Layer
       GraphLayer

Example Code:

The following code sets the range of the second data plot in layer 2 of Graph1.

```c
GraphLayer myGraph("Graph1",1);
myGraph.DataPlots(1).SetRange(5,10);
myGraph.Rescale();
```

The following sample function creates a graph from a specified template and then adds a data plot into the first layer.

```c
void myPlot()
{
    GraphPage grph;
    string strTemp = "mygrtemplate.otp";
    // Load a graph page from template
    BOOL bOK = grph.Create(strTemp, CREATE_VISIBLE);
    if (!bOK)
        return;
    GraphLayer grlay = grph.Layers(0);  //layer 1
    ASSERT(grlay.IsValid());
    Curve cv("Data1_b"); // get Curve object
    ASSERT(cv.IsValid());
    bOK = 0 <= grlay.AddPlot(cv); // add data plot
    if (!bOK)
        return;
}
```

The next example code groups the second, third, and fourth data plots in the active layer of Graph1.

```c
GraphLayer lay("Graph1");
bb = lay.GroupPlots(1, 3);
```

The following example changes the color of all the data plots in the active graph layer to the fourth color in the color list.
void setPlotColor()
{
    Project prj;
    Layer lay;
    lay = prj.ActiveLayer();
    GraphLayer glay;
    glay = (GraphLayer)lay; //explicit cast
    foreach(DataPlot dp in glay.DataPlots)
    {
        dp.SetColor(3, TRUE);
    }
}

GraphObject

Use the GraphObject class to access graphic objects on a page. Member functions include getting the name of objects, getting the number of nodes (a rectangle has four nodes), getting a node's coordinates, and extracting an object to an image file. Properties include the dimensions and coordinates of the object.

Inheritance:
OriginObject
    GraphObject

Example Code:

The following code loops through each graphic object (lines, arrows, labels, etc.) in the active layer of Graph1 using a foreach statement, and outputs the object name, its number of nodes, and the node coordinates to the current output window.
void test1()
{
    GraphLayer gl("Graph1");
    foreach(GraphObject gr in gl.GraphObjects)
    {
        double dXVal = 0, dYVal = 0;
        int iNodes = gr.GetNumOfNodes();
        if (iNodes > 1)
        {
            printf("Name = %s\n", gr.GetName());
            printf("Number of nodes = %d\n", iNodes);
            for (int ii = 0; ii < iNodes; ii++)
            {
                gr.GetNode(ii, dXVal, dYVal);
                printf("Point %d: X= %f Y = %f\n", ii, dXVal, dYVal);
            }
        }
    }
}

GraphPage

This class includes member functions to append layers from a specified graph template into the graph page, and to load a specified graph template into the graph page.

Inheritance:
OriginObject
    PageBase
        Page
            GraphPage

Example Code:

The following code creates a new graph window from a template located in the Origin folder.
GraphPage grph;
string strTemplate = "mygrtemplate.otp";
grph.Create(strTemplate);

The next example adds layers to the Graph1 window from the specified graph template.
void graph_appendlayers()
{
    GraphPage grph("Graph1");
    string strTemplate = "mygrtemplate";
    BOOL bRet = grph.AppendLayers(strTemplate);
    out_int("bRet = ", bRet);
}

Layer

This class provides functions to remove graphic objects from a layer, get and return the index of the layer, and access the colormap. It also provides a GraphObjects collection of type GraphObject which is a collection of all the graphic objects in the layer.

Inheritance:
OriginObject
    Layer

Example Code:

The following lines of code remove a text label named Text from a layer.
GraphLayer myLayer("Graph1");
myLayer.RemoveGraphObject("Text");

The next example outputs all the types of graphic objects in a layer to the current output window. It uses a foreach statement to loop through the lay.GraphObjects collection.
GraphLayer lay("Graph1"); //active layer
foreach(GraphObject go in lay.GraphObjects)
{
    ASSERT(go);
    out_str( go.GetObjectType());
}
## Layout

This class provides constructors for creating a Layout object.

**Inheritance:**

OriginObject
  - Layer
    - Layout

**Example Code:**

The following code uses a foreach statement to output all the graphic object types to the current output window.

```
Layout ly("layout1");
foreach(GraphObject go in ly.GraphObjects)
{
    ASSERT(go);
    out_str( go.GetObjectType());
}
```

## LayoutPage

This class provides constructors for creating a LayoutPage object.

**Inheritance:**

OriginObject
  - PageBase
    - Page
      - LayoutPage

**Example Code:**

The following code creates a new layout page window.

```
LayoutPage layout1;
layout1.Create();
```
matrix - 2D Dynamic Array

Origin C provides a matrix class for support of dynamic two-dimensional arrays. A matrix is not tied to an internal Origin data set and can be used with much greater flexibility. Memory allocations and destructions are handled automatically. By default, a matrix is a 2D array of double, but you can use a matrix for any basic data type. The exception is that a matrix cannot be of type string.

Inheritance:
None.

Example Code:
The following example outputs the elements of a matrix object.

```c
void myMatrixTest()
{
    matrix<int> m1={{1,2,3},{4,5,6}};
    int rows = m1.GetNumRows();
    int cols = m1.GetNumCols();
    printf("Num Rows = %d,  Num Cols = %d\n", rows, cols);
    for (int ii = 0; ii < rows; ii++)
    {
        printf("row %d:\t", ii);
        for (int jj = 0; jj < cols; jj++)
        {
            printf("%d	", m1[ii][jj]);
        }
        printf("\n");
    }
}
```
Matrix - Origin Object

Use the Matrix class to access Origin matrices.

Inheritance:
matrix
    Matrix

Example Code:

The following example creates an Origin matrix, renames the matrix, gets the number of rows and columns, and then fills the matrix with values.

```c
void fillMatrix()
{
    MatrixPage mat1;
    mat1.Create("origin");
    mat1.Rename("myMatrix");
    Matrix <double> aa;
    BOOL bRet = aa.Attach("myMatrix");
    UINT rows = aa.GetNumRows();
    UINT cols = aa.GetNumCols();
    printf("rows = %d\t cols = %d\n", rows, cols);
    for (int ii = 0; ii<rows; ii++)
        for (int jj = 0; jj<cols; jj++)
        {
            aa[ii][jj]= sin (ii) + cos (jj);
        }
}
```

As this example illustrates, after you declare a Matrix object you can access values of the matrix that are attached to that object using the following notation:

```c
objectName[rowNumber][columnNumber]
```

where:

- `rowNumber` = the row index number, starting from 0.
- `columnNumber` = the column index number, starting from 0.

MatrixLayer

Use the MatrixLayer class to access properties of a matrix window. (In Origin version 7, a matrix window can only contain one layer.) The MatrixLayer class includes functions to attach a matrix window to a MatrixLayer object, open a matrix that was saved to a file, set the number of columns.
and rows in a matrix, control image viewing, and set the data display properties. Additionally, the MatrixLayer class provides a MatrixObjects collection of type MatrixObject. These are all the MatrixObject objects in a MatrixLayer. However, in Origin 7, a MatrixLayer can only have one MatrixObject.

**Inheritance:**
OriginObject
    Layer
        Datasheet
            MatrixLayer

**Example Code:**

The following code opens a matrix that was previously saved to a file.

```c
void matrix_open()
{
    MatrixLayer mat;
    string str = "myMatrix.ogm"; // in Origin folder
    int nOption = CREATE_VISIBLESAME;
    BOOL bRet = mat.Open(str, nOption);
    out_int("bRet = ", bRet);
}
```

**MatrixObject**

Use the MatrixObject class to set the properties (Matrix Properties dialog box) of a matrix, to set the matrix dimensions, to flip the matrix vertically or horizontally, and to rotate a matrix.

**Inheritance:**
OriginObject
    DataObject
        MatrixObject

**Example Code:**

The following code sets the dimensions of the matrix1 matrix and then sets the numeric display to five significant digits.
```c
void test1()
{
    MatrixLayer ml("matrix1");
    ml.MatrixObjects(0).SetDimensions(4, 5);
    ml.MatrixObjects(0).SetDigitMode(DIGITS_SIGNIFICANT);
    ml.MatrixObjects(0).SetDigits(5);
}

The next code example flips the matrix1 matrix horizontally (first column becomes last, etc.). The code then uses a foreach statement to flip each column vertically (last cell becomes first, etc.).

```c
void test2()
{
    MatrixLayer ml("Matrix1");
    MatrixObject mo = ml.MatrixObjects(0);
    ASSERT(mo);
    mo.FlipHorizontal();
    foreach(mo in ml.MatrixObjects)
    {
        ASSERT(mo);
        mo.FlipVertical();
    }
}
```

### MatrixPage

This class provides constructors for creating a `MatrixPage` object.

**Inheritance:**

```
OriginObject
    PageBase
        Page
            MatrixPage
```

**Example Code:**

The following code creates a new matrix window.
```c
MatrixPage mat1;
// template located in Origin folder
mat1.Create("myMatrixTemplate");
```
**Note**

This class provides constructors for creating a `Note` object. It also includes a function to read and set the text in a notes window.

**Inheritance:**

```
OriginObject
    PageBase
    Note
```

**Example Code:**

The following code creates a new notes window.
```
Note myNotes;
myNotes.Create();
```

The next code example reads and sets the contents of a notes window.
```
Note myNote("Notes");
//read contents
string strNote = myNote.Text;
printf("original contents: %s\n", strNote);
//update contents
string strNew = "New contents...";
myNote.Text = strNew;
```

**OriginObject**

This is the base class for all Origin internal objects. All classes that are derived from the `OriginObject` class have access to the `OriginObject` member functions. These functions include checking the validity of an object and destroying an object.

**Page**

This is the base class for accessing layers on a page. In addition to the member functions provided to access a layer, this class provides a `Layers` collection of type `Layer` which is a collection of all the layers on the page.
Inheritance:
OriginObject
    PageBase
        Page

Example Code:

The following example function takes a Page object as an argument (must be called from another
Origin C function). It then returns the number of layers on the page to the calling function.

```c
int GetNumOfLayersInPage(Page &pp)
{
    if(pp)
        return pp.Layers.Count();
    return 0;
}
```

PageBase

This class provides member functions for creating pages, getting the name of and renaming pages, and
getting the type of a page.

Inheritance:
OriginObject
    PageBase

Example Code:

The following code creates a new graph window and then renames the window.

```c
GraphPage grph;
// template located in Origin folder
string strTemplate = "mygrtemplate.otp";
grph.Create(strTemplate);
grph.Rename("MyGraph");
```

progressBox

The progressBox class provides access to a progress bar displayed in a window, which indicates
the progress of an operation. The progress bar is a rectangle that is filled from left to right as an
operation progresses.
The `SetRange` member function controls the range of the progress bar, which is the range for the entire duration of the operation. The `Set` member function controls the current position in the operation.

**Inheritance:**
None.

**Example Code:**

The following example illustrates the use of the `progressBox` class. (Note: `LT_execute()` is a technique for including LabTalk code in your Origin C function. You might use this if the action you want to perform is not available in Origin C but is available in LabTalk.)

```c
void test(int imax = 10)
{
    progressBox aa("This is an example");
    aa.SetRange(0,imax);

    for (int ii = 0; ii < imax; ii++)
    {
        if (aa.Set(ii))
            printf("Hello, at %d\n", ii);
        else
        {
            out_str("User abort!");
            break;
        }

        LT_execute("sec -p 0.5"); // wasting some time
    }
}
```

**Project**

The `Project` class represents the current project in Origin. This class provides collections for all the windows in a project, and for all the graph, worksheet, notes, layout page, and matrix windows in a project. It also provides functions to access particular windows by index or by name, or particular graph, worksheet, notes, layout page, and matrix windows by index or by name.

**Inheritance:**
None.
Example Code:

The following example illustrates the use of the foreach statement and the collection of pages (windows) in the project.

```c
Project prj;
PageBase pg;
Collection<PageBase> pgcoll;
// initialize collection object from Pages property of Project class:
pgcoll = prj.Pages;
// loop over all the pages in the collection:
foreach (pg in pgcoll)
{
    string strName = pg.GetName();
    out_str(strName);
}
```

**ROIObject**

This class is currently not in use.

**stdioFile**

Origin C provides the file and the stdioFile classes for reading and writing to files. The Origin C file class has nearly the same functionality as the CFile class documented in the Microsoft Foundation Class Library. Similarly, the Origin C stdioFile class provides nearly the same functionality as the CStdioFile class. The stdioFile class provides buffered access to the file and is thus faster than the file class, which uses a handle.

**Inheritance:**

file
    stdioFile

**Example Code:**

The following code gets the Origin folder path and creates a new standard I/O file.

```c
string strOriginPath;
strOriginPath = GetAppPath() + "OriginC";
strFile = strOriginPath + "\test.txt";
stdioFile ff(strFile, file::modeCreate | file::modeReadWrite);
```
The **string** class provides support for character strings. This class is modeled after CString in the Microsoft Foundation Class Library.

**Inheritance:**
None.

**Example Code:**

In the following example, the `getStrTest` function returns a `string` to the calling function.

```c
string getStrTest();
void myTest()
{
    printf("%s\n",getStrTest());
}

string getStrTest()
{
    string strB;
    double ff = PI;
    strB.Format("The value of PI is %f",ff);
    return strB;
}
```

The next example accesses a token in a string and outputs it to the current output window.

```c
void getTokens()
{
    string str1 = "How are you?";
    string str = str1.GetToken(2);
    out_str(str); // output you?
}
```

The next example accesses elements in a string.

```c
void getElements()
{
    string str("test");
    int ii = str[1]; // ASCII code for e is 101
    str[2] = 43; // 43 is ASCII code for +
    str[3] = str[2];
    printf(str); // output te++
}
You can also declare a vector object of type string for support of dynamic one-dimensional arrays. Origin C includes a typedef in string.h to declare a new data type name StringArray.

typedef vector<string> StringArray;

The following example illustrates the use of StringArray.

```c
void displayContents()
{
    StringArray myString;
    string strTemp = "Mon Tue Wed Thu Fri Sat Sun";
    int nCount = strTemp.GetTokens(myString);
    for (int ii = 0; ii < nCount; ii++)
    {
        printf(myString[ii]);
        printf("\n");
    }
}
```

**StyleHolder**

Use the StyleHolder class to get the plot ID, plot type, and data set plotting designations (X, Y, etc.) for the specified data plot(s).

*Inheritance:*

OriginObject
   GraphObject
      StyleHolder

*Example Code:*

The following code outputs the data plot type for the first data plot in layer 1 of Graph1 to the current output window.
void plotdescr()
{
    GraphPage grph("Graph1");
    GraphLayer lay = grph.Layers(0);
    if(lay)
    {
        StyleHolder sh;
        sh = lay.StyleHolders(0);
        out_str(sh.GetDescription());
    }
}

The next example outputs the plot ID for each of the data plots in layer 1 of Graph1 to the current output window.
void plotIDs()
{
    GraphPage grph("Graph1");
    GraphLayer lay = grph.Layers(0);
    foreach(StyleHolder sh in lay.StyleHolders)
    {
        out_int("plot type is ", sh.GetPlotId());
    }
}

This next example is identical to the previous one except it outputs the data set plotting designations for each data plot.
void plotDesignation()
{
    GraphPage grph("Graph1");
    GraphLayer lay = grph.Layers(0);
    foreach(StyleHolder sh in lay.StyleHolders)
    {
        out_str(sh.GetPlotDesignations());
    }
}

vector

Origin C provides a vector class for support of dynamic one-dimensional arrays. A vector is not tied to an internal Origin data set and can be used with much greater flexibility. Memory allocations
and destructions are handled automatically. By default, a `vector` is a 1D array of `double`, but you can use a `vector` for any basic data type.

Note: `StringArray` is a synonym for a `vector` of type `string`.

*Inheritance:*
None.

*Example Code:*

The following example illustrates how to declare and assign values to vectors. It also illustrates how you can perform arithmetic operations on vectors, and it shows how you can cast a `double` vector into an `int` vector.

```c
void test(int imax = 10)
{
    vector<double> aa(imax);
    vector bb(imax); // same as aa, also double
    vector <int> akk;
    for (int ii = 0; ii < aa.GetSize(); ii++)
        aa[ii] = rnd();
    bb = 10*aa; // vector arithmetic like scalers
    // can cast double vector into int vector
    akk = bb;
    ASSERT(akk.GetSize() == imax);
    for (ii = 0; ii < akk.GetSize(); ii++)
        printf("akk[%d]=%d after cast from %f\n",ii,akk[ii],bb[ii]);
}
```

**waitCursor**

The `waitCursor` class provides access to an hourglass wait cursor that you can display when an operation is in progress. To display the hourglass, declare the object before the code that performs the lengthy operation. The object's destructor automatically changes the cursor back to its default display at the end of the block in which the object is declared.

*Inheritance:*
None.

*Example Code:*

The following example illustrates the use of the `waitCursor` class.
void test(int imax = 100000)
{
    waitCursor wait;
    double aa = 0.5;
    DWORD dw1 = GetTickCount();
    for (int ii = 0; ii < imax; ii++)
    {
        aa = sin(aa) + ii * aa;
        printf("It took %d ms for the %d loop\n", GetTickCount()-dw1, imax);
    }
}

Worksheet

The Worksheet class includes functions to attach an Origin worksheet to a Worksheet object, open a worksheet that was saved to a file, perform column and row operations such as adding and deleting columns and rows, perform cell operations, copy the worksheet contents into a specified output window, and paste data into a worksheet. Additionally, the Worksheet class provides a Columns collection of type Column which is a collection of all the columns in the worksheet. You can thus use a foreach statement to loop through all columns in the worksheet (see example).

Inheritance:

OriginObject
    Layer
        DataSheet
            Worksheet

Example Code:

The following code creates a new worksheet and renames it from the argument passed in the function call. This example then adds two columns to the worksheet and renames them.
void myWorksheet(string winName)
{
    //Create a worksheet named winName
    Worksheet wks;
    wks.Create("Origin.otw", CREATE_VISIBLESAME);
    wks.GetPage().Rename(winName);
    //Add two columns to the Worksheet wks
    string strColName;
    int colNum = wks.AddCol("MyFirst", strColName);
    wks.AddCol("MySecond", strColName);
}
In the following example, all the columns in the Data1 worksheet are set to Numeric Display = Significant Digits with the number of digits equal to 10. In the foreach statement, wks.Columns is a collection of all the columns in the Data1 worksheet.

```c
void setMyCols()
{
    Worksheet wks("data1");
    foreach(Column cols in wks.Columns)
    {
        cols.SetDigitMode(DIGITS_SIGNIFICANT);
        cols.SetDigits(10);
    }
}
```

**WorksheetPage**

This class provides constructors for creating a WorksheetPage object.

**Inheritance:**
OriginObject
    PageBase
        Page
            WorksheetPage

**Example Code:**

The following code creates a new worksheet window.

```c
WorksheetPage wks;
int nOption = CREATE_VISIBLE_SAME;
BOOL bRet = wks.Create("origin", nOption);
```
Code Examples

Basic I/O

Origin C supports the following global functions for input and output:

BrowseGetPath (Only available in version 7 SR1 and later. Defined in sys_utils.c, which is loaded automatically.)

GetOpenBox (Only available in version 7 SR1 and later. Defined in sys_utils.c, which is loaded automatically.)

InputBox (prototyped in utilities.h)
out_string (prototyped in stdio.h)
out_int (prototyped in stdio.h)
out_double (prototyped in stdio.h)
out_complex (prototyped in complex.h)
printf (prototyped in stdio.h)
MessageBox (prototyped in mswin.h)

The following code example illustrates the use of GetOpenBox to open the Open dialog box for a single type of file.

```c
string strFileName = GetOpenBox("*.ocw Workspace", GetAppPath() + "OriginC\");  
if(strFileName.IsEmpty())
    out_str("User has cancelled the Open dialog");
else
    printf("The file selected is %s\n", strFileName);
```

The next code example illustrates the use of InputBox.

```c
string strName = InputBox("Enter your name");
out_str(strName);
```

The following function illustrates the use of printf.

```c
void basicIO()
{
    double vv = cos(3.14);
    int nn = 45;
    String str = "Test";
    printf("The string = %s, nn = %d, vv = %f\n", str, nn, vv);
}
```
The following function illustrates the use of `out_double`.

```c
void dd()
{
    double a = 5e-5;
    out_double("the value is ",a);
}
```

Origin C also provides classes for I/O support. These include the `file` and `stdioFile` classes for reading and writing to files, and the `string` class for manipulating and outputting strings.

## String Support

Origin C supports a `string` data type, similar to `CString` in Visual C. In the following example, the `getStrTest` function returns a `string` to the calling function.

```c
string getStrTest();

void myTest()
{
    printf("%s\n",getStrTest());
}

string getStrTest()
{
    string strB;
    double ff = PI;
    strB.Format("The value of PI is %f",ff);
    return strB;
}
```

The next example accesses a token in a string.

```c
void getTokens()
{
    string str1 = "How are you?";
    string str = str1.GetToken(2);
    out_str(str); // output you?
}
```
The next example accesses elements in a string.

```c
void getElements()
{
    string str("test");
    int ii = str[1]; // ASCII code for e is 101
    str[2] = 43; // 43 is ASCII code for +
    str[3]= str[2];
    printf(str); // output te++
}
```

You can also declare a `vector` object of type `string` for support of dynamic arrays. Origin C includes a `typedef` in `string.h` to declare a new data type name `StringArray`.

```c
typedef vector<string> StringArray;
```

The following example illustrates the use of `StringArray`.

```c
void displayContents()
{
    StringArray myString;
    string strTemp = "Mon Tue Wed Thu Fri Sat Sun";
    int nCount = strTemp.GetTokens(myString);
    for (int ii = 0; ii < nCount; ii++)
    {
        printf(myString[ii]);
        printf("\n");
    }
}
```

**Dynamic One and Two-dimensional Arrays**

Origin C provides `vector` and `matrix` classes for support of dynamic one-dimensional and two-dimensional arrays. A `vector` and `matrix` are not tied to an internal Origin data set and can be used with much greater flexibility. Memory allocations and destructions are handled automatically. By default, a `vector` is an array of `double` and a `matrix` is a 2D array of `double`, but you can use a `vector` or `matrix` for any basic data type. The exception is that a `vector` can be of type `string` but a `matrix` can't. As discussed in the previous section, `StringArray` is a synonym for a `vector` of type `string`. 

---

**Code Examples • 41**
void test(int imax = 10)
{
    vector<double> aa(imax);
    vector bb(imax); // same as aa, also double
    vector<int> akk;
    for (int ii = 0; ii < aa.GetSize(); ii++)
        aa[ii] = rnd();
    bb = 10*aa; // vectors can have arithmetics just like scalers
    akk = bb; // can cast double vector into int vector
    ASSERT(akk.GetSize() == imax);
    for (ii = 0; ii < akk.GetSize(); ii++)
        printf("akk[%d]=%d after cast from %f\n", ii, akk[ii], bb[ii]);
}

The following example outputs the elements of a matrix object.

void myMatrixTest()
{
    matrix<int> m1={{1,2,3},{4,5,6}};
    int rows = m1.GetNumRows();
    int cols = m1.GetNumCols();
    printf("Num Rows = %d,  Num Cols = %d\n", rows, cols);
    for (int ii = 0; ii < rows; ii++)
    {
        printf("row %d:\t", ii);
        for (int jj = 0; jj < cols; jj++)
        {
            printf("%d\t", m1[ii][jj]);
        }
    }
}
Complex Numbers

Origin C also has support for complex numbers, as shown in the following example:

```c
void test_complex()
{
    complex a = 3.4+5.6i;
    complex b = sin(a) + 1i; // must use 1i instead of just i;
    double aa = Re(b);
    double bb = aa + a * b;
}
```

Using Origin C for Data Acquisition

RS-232 Operation

The following example shows you how to send command instructions and receive command responses over the COM1 serial port using Origin C. For details, Origin 7 customers can refer to the Origin C sample file rs232.c which is located in your Origin\OriginC\OriginLab subfolder.

1) Include the mscomm.h header file at the top of your Origin C source file. This header file contains Win32 serial communication function prototypes.

```c
#include <mscomm.h>
```

2) Define a global file handle variable for COM1 which will be used for reading and writing operations.

```c
file ffComm;
```

3) Open and set up the RS-232 port.

```c
// Open com port COM1
if (!ffComm.Open("COM1:", file::modeReadWrite ))
{
    ASSERT(FALSE);
    return FALSE;
}
```

```c
UINT hCom = ffComm.GetHandle();
if (hCom == file::hFileNull )
{
    ASSERT(FALSE);
    return FALSE;
}
```
// Fill in a DCB structure with the current configuration, 
// then the DCB structure is modified to reconfigure the serial port.

DCB dcb;
if (!GetCommState((HANDLE)hCom, &dcb))
{
    ASSERT(FALSE);
    return FALSE;
}

// dcb parameters for user
dcb.BaudRate = BAUD_9600;  // set the baud rate
dcb.ByteSize = 8;  // data size
dcb.Parity   = NOPARITY;  // no parity bit
dcb.StopBits = ONESTOPBIT;  // one stop bit

// dcb fixed parameters
dcb.fBinary  = 1;
dcb.fParity  = 0;
dcb.fOutxCtsFlow = 0;
dcb.fOutxDsrFlow = 0;
dcb.fDtrControl = 0;
dcb.fDsrSensitivity = 0;
dcb.fTXContinueOnXoff = 0;
dcb.fRtsControl = 0;

if (!SetCommState((HANDLE)hCom, &dcb))
{
    ASSERT(FALSE);
    return FALSE;
}

4) Set up the time-out parameters for the RS-232 serial port. The following sample code shows how to use the COMMTIMEOUTS structure in the SetCommTimeouts function to set and query the time-out parameters for the RS-232 serial port. The parameters will determine the behavior of Read and Write operations on the port.
UINT hCom = ffComm.GetHandle();
if ( hCom == file::hFileNull )
{
    ASSERT(FALSE);
    return FALSE;
}

COMMTIMEOUTS tTimeout;

    tTimeout.ReadIntervalTimeout         = MAXWORD;
    tTimeout.ReadTotalTimeoutMultiplier = 0;
    tTimeout.ReadTotalTimeoutConstant   = 5000; // 5 S
    tTimeout.WriteTotalTimeoutMultiplier = 0;
    tTimeout.WriteTotalTimeoutConstant  = 0;

    // config the timeout
    if( !SetCommTimeouts((HANDLE)hCom,&tTimeout) )
    {
        ASSERT(FALSE);
        return FALSE;
    }

5) Write to the RS-232 port.
   string strCommand = "Instruction"

   UINT hCom = ffComm.GetHandle();
   if (hCom == file::hFileNull )
   {
       ASSERT(FALSE);
       return FALSE;
   }

   int nLength = strCommand.GetLength();
   DWORD dwWrite = ffComm.Write(strCommand.GetBuffer(nLength), nLength
   );

   printf("%d bytes chars have been sent COM.\n", dwWrite );
6) Read from the RS-232 port.
#define BUFFER_BYTES 256
char lpBuff[BUFFER_BYTES];

DWORD dwRead = 0;
DWORD dwTotal = 0;

string strTotal = "";

// Read loop for com port.
while ( dwRead = ffComm.Read( lpBuff, BUFFER_BYTES - 1 ) )
{
    dwTotal += dwRead;
    string str(lpBuff, dwRead);
    strTotal += str;
}

printf("%d bytes chars have been read:\n", dwTotal);
printf( strTotal );

7) Close the handle for COM1.
if( !ffComm.Close() )
{
    printf("Error when close COM.\n");
    return FALSE;
}

**GPIB(General Purpose Interface Bus)Operation**

GPIB is a standard interface for communication between instruments and controllers from various vendors. The following example shows you how to use Origin C to operate on an HP34401A HP-IB port via NI-488.2™ GPIB DLL. For details, Origin 7 customers can refer to the Origin C sample file GPIB.c which is located in your Origin (SR1) \OriginC\OriginLab subfolder.

1) Install the NI488.2 driver on your computer. Origin C will access its DLL to operate on the GPIB card.

2) Include the NI488-2.h header file which contains constants and prototypes for GPIB-32.dll.
#include <NI488-2.h>

Note that the syntax #pragma dll(gpiib-32, system) was used in this file to specify the gpiib-32.dll, which should already be located in your Windows System folder.
3) Define two global variables at the beginning of your Origin C file.

```c
// Global variable for the GPIB card
int Device = 0;  // Device unit descriptor
int BoardIndex = 0;  // Interface Index (GPIB0=0,GPIB1=1,etc.)
```

4) Initialize the GPIB device. Using the `ibdev` function will bring the device online. A device handle is returned and will be used in all subsequent calls to the device.

```c
int PrimaryAddress = 8;  // The Primary address of device
// It can be read from NI GPIB explorer
int SecondaryAddress = 0;  // Secondary address

char Buffer[21];  // Read buffer

Device = ibdev(  // Create a unit descriptor handle
    BoardIndex,  // Board Index (GPIB0 = 0)
    PrimaryAddress,  // Device primary address 8
    SecondaryAddress,  // No secondary address
    T10s,  // Timeout setting (T10s = 10 seconds)
    1,  // Assert EOI line at end of write
    0);  // EOS termination mode
```

5) Write to the GPIB port. Use the `ibwrt` function to write some instructions to the GPIB port.

```c
ibclr(Device);  // First Clear the device

char beep[] = "SYST:BEEP";
ibwrt(Device, beep, sizeof(beep));

char disp[] = "DISP:TEXT 'ORIGIN C'"; // Textout Origin C
ibwrt(Device, disp, sizeof(disp));

Sleep(1000);  // Delay 1 s

char rst[] = "*RST";  // Reset
ibwrt(Device, rst, sizeof(rst));

char cls[] = "*CLS";  // Clear register
ibwrt(Device, cls, sizeof(cls));

char meas[] = "CONF:VOLT:DC 10";  // Measure Mode : DC 1v range
ibwrt(Device, meas, sizeof(meas));
```
char coun[] = "SAMP:COUN 1";  // Data Sample counts
ibwrt(Device, coun, sizeof(coun));

6) Read from the GPIB port. Use the ibrd function to read from the GPIB port. The following code shows how to read and put the results into an Origin C vector.
vector<double> data(20);  // Define double type vector.

char  Buffer[21];  // read buffer, set size as 21

for( int ii = 0; ii < data.GetSize(); ii++)
{
    char read[] = "READ?";  // Read query command
    ibwrt(Device, read, sizeof(read));

    ibrd(Device, Buffer, 20);  // Read up to 20 bytes from the device

    printf(Buffer);
    Buffer[user_ibcntl] = '\0';  // Null terminate the ASCII string
    data[ii] = atof(Buffer);
    Sleep(100)  // 0.1 second
}

7) Do some clean up work. Use the ibonl function to take both the device and the interface offline.
ibonl(Device, 0);  // Take the device offline
ibonl(BoardIndex, 0);  // Take the interface offline

---

**Calling FORTRAN Routines from Origin C**

Origin C can call functions exported from Fortran, allowing you to use your existing Fortran code from Origin C. The Fortran routines can call functions from existing libraries of mathematical and statistical functions, such as IMSL. The calling, naming, and argument passing conventions between Fortran and Origin C are the same as between Fortran and other standard ANSI C compilers.

To call Fortran from Origin C, perform the following steps:

1) Create a Fortran DLL with your Fortran code. Export the routines that you want to make available to Origin C using the using the !DEC$ ATTRIBUTES DLLEXPORT compiler directive.

2) In a header file in Origin C, use #pragma dll(dllName) to load the DLL.

3) Prototype the Fortran routines using the __stdcall keyword in your Origin C header file.
4) When passing arguments between Origin C and Fortran, make sure that the calling program and the called routine agree on how each argument is passed. By default, arguments in Origin C are passed by value (like in C), whereas in Fortran they are passed by reference. However, you can use the ATTRIBUTES option to tell Fortran how an argument is passed (see Examples). You can pass arrays as arguments as well (and therefore Origin data sets and matrices). However, when passing 2-dimensional arrays, keep in mind that in Fortran they are stored “column first”, whereas in Origin C they are stored by row. So they must be “reordered” before and after passing to Fortran (see Examples).

**Example 1**

This example calls a FORTRAN function `FSUM` to find the sum of two numbers. Arguments are passed by reference.

Origin C code:
```c
int a, b, n;
int n = FSUM(&a, &b);
```

Fortran function:
```fortran
INTEGER(4) FUNCTION INTSUM(N, M)
!DEC$ ATTRIBUTES DLLEXPORT:: INTSUM ! procedures is being exported to another application
INTEGER N, M
   INTSUM = N + M
END
```

Fortran function prototype in Origin C header file:
```c
int __stdcall INTSUM(int* N, int* M);
```

**Example 2**

This example calls a FORTRAN function `FSUM` to find the sum of two numbers. Arguments are passed by value.

Origin C code:
```c
double a, b;
double y;
y = SUMFT(a, b);
```
Fortran function:
REAL(8) FUNCTION SUMFT(A, B)
!DEC$ ATTRIBUTES DLLEXPORT:: SUMFT ! procedures is being exported
!DEC$ ATTRIBUTES VALUE :: A ! This argument is passed by value
!DEC$ ATTRIBUTES VALUE :: B ! This argument is passed by value
REAL(8) A, B
    SUMFT = A + B
END

Fortran function prototype in Origin C header file:
double __stdcall SUMFT(double A, double B);

Example 3
This example computes the number of elements in a data set. The data set is passed to Fortran as a one-dimensional array.

Origin C code:
void dd(string strWksName = "Data1")
{
    Dataset data(strWksName,0); //col 1 of wks
    vector<float> vv; vv = data;
    int n = vv.GetSize();
    float ave = CALC_SUM(vv,&n);
}

Fortran function:
REAL(4)  FUNCTION CALC_SUM(D, N)
!DEC$ ATTRIBUTES DLLEXPORT:: CALC_SUM ! procedures is exported
REAL(4) D
DIMENSION D(N)
INTEGER N
    CALC_SUM = SUM(D)
END FUNCTION CALC_SUM

Fortran function prototype in Origin C header file:
float __stdcall CALC_SUM(float* D, int* N);

Example 4
This example computes the inverse of a matrix. It shows how to pass a two-dimensional array to Fortran. The Fortran routine calls an IMSL function to compute the inverse.
Origin C code:
void mmInvert(string strMatrix = "Matrix1")
{
    Matrix mat(strMatrix);
    matrix<float> fm;
    fm = mat;
    int nRows = mat.GetNumRows();
    int nCols = mat.GetNumCols();
    if (nRows != nCols)
    {
        printf("can only invert a square matrix");
        return;
    }

    //set up the output matrix – an actual Origin matrix is created
    MatrixLayer Matrix2;
    Matrix2.Create();
    Matrix2.SetNumRows(nCols);
    Matrix2.SetNumCols(nRows);
    Matrix2.SetInternalData(FSI_REAL);
    matrix<float> outm(Matrix2);

    //reorder the input matrix before passing to FORTRAN routine (in FORTRAN
    //matrix elements are stored by column, but in C they are by row
    reorder(fm);
    //call FORTRAN routine to transpose matrix; this FORTRAN routine will
    //in turn call an IMSL routine
    INVERT(&nRows, fm, outm);
    reorder1(outm);  //reorder the output matrix
}
Fortran function:

```fortran
SUBROUTINE INVERT(N, A, INVA)
!DEC$ ATTRIBUTES DLLEXPORT:: INVERT
!Input args
INTEGER N
REAL A(N,N)
!Output argument
REAL INVA(N,N)
!Call IMSL function
CALL LINRG (N, A, N, INVA, N)
RETURN
END
```

Fortran function prototype in Origin C header file:

```c
void __stdcall INVERT(int* N,float* A,float* INVA);
```

## Importing Data

Origin C provides the `file` and the `stdioFile` classes for reading and writing to files. The Origin C `file` class has nearly the same functionality as the `CFile` class documented in the Microsoft Foundation Class Library. Similarly, the Origin C `stdioFile` class provides nearly the same functionality as the `CStdioFile` class. The `stdioFile` class provides buffered access to the file and is thus faster than the `file` class, which uses a handle.

The following example functions illustrate how you can use the `file` class to export and import worksheet data to a binary file. In this example, a binary file is designed with a simple structure, namely a file header (see `struct MyMainHeader`) followed by each column of data stored as 2 byte (`short`) integers with the first four byte (`long`) for the number of points in that column. Data in Origin worksheets are `double` values, so a calibration factor is used to convert these integers (`MyMainHeader.factor`).

```c
#include <origin.h>
// the header for this example is binfile.h
// header code is listed after functions
#include "binfile.h"
// copy data from given column to a short vector
// by dividing each point by a factor
// return number of points
UINT ColumnDataToIntVector(Column& cc, vector<short>& vv, float factor)
{
    Dataset aa(cc);
    UINT nSize = aa.GetSize();
    vv.SetSize(nSize);
```
for(int ii = 0; ii < nSize; ii ++)
    vv[ii] = nint(aa[ii]/factor);
return nSize;
}

// copy data from a short vector to a column
// return number of
BOOL IntVectorToColumnData(Column& cc, vector&lt;short&gt;& vv, float factor)
{
    Dataset aa(cc);
    UINT nSize = vv.GetSize();
    aa.SetSize(nSize);
    for(int ii = 0; ii < nSize; ii ++)
        aa[ii] = factor * vv[ii];
    return TRUE;// worry about error checking later
}

// strFileName must be supplied
// strWksName = "" to use the current window
// return TRUE for success, FALSE for error and
// error messages are printed to Script window
BOOL myImport(string strFileName, string strWksName)
{
    Worksheet wks;
    if(strWksName.IsEmpty())
        //explicit cast:
        wks = (Worksheet)Project.ActiveLayer();
    else
        wks.Attach(strWksName);
    if(wks==NULL)
    {
        out_str("invalid worksheet");
        return FALSE;
    }
    file myFile;
    if(myFile.Open(strFileName, file::modeRead) == FALSE)
    {
        printf("Cannot open file %f for reading\n", strFileName);
        return FALSE;
    }
    MyMainHeader myHeader;
    // read file header, do simple test for validity
if(myFile.Read(&myHeader, sizeof(MyMainHeader)) !=
sizeof(MyMainHeader) || myHeader.nColumns <= 0 || myHeader.nColumns >
100) // assume <= 100 cols
{
    printf("File %s is empty or not created by the myExport
    function\n", strFileName);
    return FALSE;
}

//-------------------------------
// remove all current cols and create new ones
// remove all cols
while(wks.DeleteCol(0)) // remove all cols
{
    int ii;
    for(ii = 0; ii < myHeader.nColumns; ii++)
        wks.AddCol();
    vector<short> vTemp;
    UINT nSize;
    Column cc;
    for(ii = 0; ii < myHeader.nColumns; ii++)
    {
        cc = wks.Columns(ii);
        if(myFile.Read(&nSize, sizeof(nSize)) != sizeof(nSize))
            return FALSE;
        vTemp.SetSize(nSize);
        if(myFile.Read(vTemp, sizeof(short)*nSize) !=
sizeof(short)*nSize)
            return FALSE;
        IntVectorToColumnData(cc, vTemp, myHeader.factor);
    }
    return TRUE;
}

// strFileName must be supplied
// strWksName = "" to use the current window
// return TRUE for success, FALSE for error and
// error messages are printed to Script window
BOOL myExport(string strFileName, string strWksName)
{
    Worksheet wks;
    if(strWksName.IsEmpty())
        wks = (Worksheet)Project.ActiveLayer();
    else
wks.Attach(strWksName);
if(wks==NULL)
{
    out_str("invalid worksheet");
    return FALSE;
}

file myFile;
if(myFile.Open(strFileName, file::modeCreate | file::modeWrite) == FALSE)
{
    printf("Cannot create file %f for writing\n", strFileName);
    return FALSE;
}

MyMainHeader myHeader;
lstrcpyn(myHeader.szTitle, wks.GetPage().Label, MY_TEXT_SIZE);
myHeader.factor = 3.0 / 2048.0;
// example for 11 bit to map to 3 volts
myHeader.nColumns = wks.GetNumCols();
myFile.Write(&myHeader, sizeof(MyMainHeader));
vector<short> vTemp;
UINT nSize;
Column cc;
for(int ii = 0; ii < myHeader.nColumns; ii++)
{
    cc = wks.Columns(ii);
    nSize = ColumnDataToIntVector(cc, vTemp, myHeader.factor);
    myFile.Write(&nSize, sizeof(UINT));
    myFile.Write(vTemp, sizeof(short)*nSize);
}
return TRUE;
}

The header for this example is binfile.h. This file contains the following code.

#ifndef _BINFILE_H
#define _BINFILE_H
#define MY_TEXT_SIZE 100

typedef struct tagMyMainHeader
{
    char szTitle[MY_TEXT_SIZE];
    short nColumns;
}
float factor; // calibration constant
}MyMainHeader;

BOOL myImport(string strFileName, string strWksName = "");
BOOL myExport(string strFileName, string strWksName = "");

#endif // _BINFILE_H

Creating and Accessing Data Sets

The following Origin C function illustrates how you can use the Worksheet, Column, and Dataset classes to create data sets.

```c
void wksTest(string winName)
{
    Worksheet wks;
    wks.Create("Origin.otw", CREATE_VISIBLE_SAME);
    wks.GetPage().Rename(winName);
    string strColName;
    int colNum = wks.AddCol("MyX", strColName);
    wks.AddCol("MyY", strColName);
    UINT rows = wks.GetNumRows();
    Column cc(winName, colNum);
    cc.SetType(OKDATAOBJ_DESIGNATION_X);
    Dataset dd1(winName + ":MyX");
    Dataset dd2(winName + ":MyY");
    dd1.SetSize(rows);
    dd2.SetSize(rows);
    for (int ii = 0; ii < rows; ii++)
    {
        dd1[ii] = ii;
        dd2[ii] = sin(ii);
    }
}
```

You could then call this function from Origin using the following LabTalk command.

`wksTest myNewWorksheet;`

As this example illustrates, after you declare a Dataset object you can access values of the data set that are attached to that object using the following notation:

`ObjectName[ElementNumber]`

where `ElementNumber` starts at 0 (which contrasts with LabTalk, in which the index starts at 1).
Performing Math Operations on Data Sets

You can perform calculations on data sets in Origin C by assigning the result of an expression involving Dataset objects to a Dataset. For example, the following lines of Origin C code add the row by row elements of two data sets together with the results placed in a third data set.

```origin_c
Dataset dd1("data1_b");
Dataset dd2("data1_c");
Dataset dd3("data1_d");
dd3 = dd1 + dd2;
```

When you perform a Dataset calculation like this, the data sets must be the same size or you will get a run time error. To get and set the size of a Dataset, use the inherited GetSize and SetSize member functions of the vector class.

Plotting Data

The following code example illustrates how to create a graph window from a template, add a data plot to a layer, add an additional layer, and then add another data plot to the new layer.

```origin_c
void an_example()
{
    GraphPage grph;
    string strTemplate = "mytemplate.otp";
    BOOL bOK = grph.Create(strTemplate, CREATE_VISIBLE);
    if (!bOK)
        return;
    // Get the first layer:
    GraphLayer grlay = grph.Layers(0);
    ASSERT(grlay.IsValid());
    // Get the curve object from "Data1_b" data set
    Curve cv("Data1_b");
    ASSERT(cv.IsValid());
    // Add one data plot to the graph layer:
    bOK = 0 <= grlay.AddPlot(cv);
    if (!bOK)
        return;
    // Rescale layer:
    grlay.Rescale();
    // Add another layer to graph page:
    bOK = grph.AppendLayers(NULL);
    if (!bOK)
        return;
}```
// Get the second layer:
GraphLayer grlay2 = grph.Layers(1);
ASSERT(grlay2.IsValid());
// Get the curve object from "Data1_c" data set
Curve cv2("Data1_c");
ASSERT(cv2.IsValid());
// Add one data plot to the second layer:
bOK = 0 <= grlay2.AddPlot(cv2);
// Rescale layer:
grlay.Rescale();
}

Nonlinear Curve Fitting

Origin C provides global functions for performing nonlinear curve fitting. In the following example function, a gaussian fit is performed on the data set that is passed to the function.

```c
void GaussianFit(string strCurve)
{
    // create an NLSFcntrl structure variable
    // see OC_Types.h for structure definition
    NLSFCntrl control;
    // initialize structure
    // see internal.C for function
    initNLSF(control);
    // initialize dependent variable
    lstrcpy(control.szYdataName, strCurve);
    // initialize fitting function
    lstrcpy(control.szFuncName, "gaussamp");
    // perform fit, see internal.C for function
    fitNLSF(control);
}
```

You can call this function from Origin using the following LabTalk command. The %c system variable holds the name of the active Y data set.

`GaussianFit(%c);`

In addition to calling these global nonlinear fitting functions, you can also call the NLSF (fitter) functions from Origin C. This includes all the built-in functions available in the NLSF.

There are two ways to call an NLSF function from your Origin C function:

1) In the first form, you use the prefix nlfx before the function name. The general form is:
\[ y = \text{nlf} \text{FuncName}(arguments); \]

\text{FuncName} is the function name taken from the following line in the NLSF function's FDF file:

\text{Function Source = fgroup.FuncName}

The argument list for the function is listed in the [FITTING PARAMETERS] section FDF file. When you call the NLSF function from Origin C, you must list the arguments in the same order as they appear in the FDF file. Also, you should insert an initial argument which is the \text{x} value.

For example, the general form you would use to call the Gauss NLSF function is:

\[ y = \text{nlfGauss}(x, y0, xc, w, A); \]

Keep in mind that all the NLSF parameters are of type \text{double}, and the function returns a \text{double}.

The following code example illustrates the use of this function form.

\begin{verbatim}
void test1()
{
    double y0, xc, w, A;
    double x, y1;
    y0 = 5;
    xc = 2;
    w = 1;
    A = 0.5;
    x = 1.8;
    y1 = \text{nlfGauss}(x, y0, xc, w, A);
    \text{out_double}("y1 = ", y1);
}
\end{verbatim}

2) In the second form, you use the prefix \text{nlf} (no \text{x}) before the function name. The general form is:

\[ y = \text{nlfFuncName}(x, arrayName, numElements); \]

\text{FuncName} is the same as described in the first form. However, in this second form, you pass the function the \text{X} value, an array, and the number of elements in the array. The array elements are the function parameters as listed in the function's FDF file.

The following code example illustrates the use of this function form.

\begin{verbatim}
void test2()
{
    double aa[4];
    double y0, xc, w, A;
    double x, y2;
    y0 = 5;
    xc = 2;
    w = 1;
    A = 0.5;
    x = 1.8;
    aa[0] = y0;
    aa[1] = xc;
}
\end{verbatim}
aa[2] = w;
y2 = nlfGauss(x, aa, 4);
out_double("y2 = ", y2);
}

Integration

Origin C provides a Curve_integrate global function to perform an integration on the given Curve. By default, the integration is performed from Y = 0, although you can optionally specify a baseline to integrate from.

In the following example, integration is performed on the specified data sets and the area and peak results are output.

```c
void myIntegrate(string strDataX, string strDataY)
{
    Curve cc(strDataX, strDataY);
    IntegrationResult result;
    Curve_integrate(&cc, &result);
    printf("The area is %.1f\n", result.Area);
    printf("The peak position is %d\n", result.iPeak);
}
```

You can call this function from Origin using the following LabTalk command, where the arguments are an X and Y data set.

```labtalk
myintegrate(gaussian_pos, gaussian_ampl);
```

Differentiation

Origin C provides a Curve_derivative global function to perform a differentiation on the given Curve. This function is prototyped in data.h as follows:

```c
BOOL Curve_derivative(Curve* pData, Curve* pResults, int nOrder, double dSmoothness = LET_THE_PROGRAM_CHOOSE, Dataset* pWeights = NULL);
```

The following code example performs a differentiation on the data in the first and second column of the specified worksheet. Two new columns are added to the worksheet and the derivative results are added to these columns.
void test(string wksName, int colX, int colY)
{
    Worksheet wks(wksName);
    Curve cc1(wks, colX, colY);
    int newColX = wks.AddCol("DerivX");
    int newColY = wks.AddCol("DerivY");
    wks.Columns(newColX).SetType(OKDATAOBJ_DESIGNATION_X);
    Curve cc2(wks, newColX, newColY);
    Curve_derivative(&cc1, &cc2, 2);
}

You can call this function from Origin using the following LabTalk command:

test(data1, 0, 1);

Linear and Polynomial Regression

Origin C provides two global functions for performing polynomial regression (set the order to 1 for linear regression). These functions are prototyped in data.h as follows:

`BOOL fitpoly(Curve &cc, int ipolyorder, double *coeff, int inumer = 0, int idenom = 0);`

`BOOL fitpoly_range(Curve &cc, int ibeg, int iend, int ipolyorder, double *coeff);`

The following code example illustrates the use of the `fitpoly` function. A Curve object is created and attached to `Data1_A` and `Data1_B`. Then an array is declared to hold the polynomial coefficients from the fit. The array is initialized for four members, as a 3rd order polynomial fit will be performed. The `fitpoly` function is called and passed the Curve, the order, and the array. The array members are then output to the current output window.

```c
void PolyFit1()
{
    Curve crvMyData("Data1_A", "Data1_B");
    double dCoeff[4];
    bool bRet = fitpoly(crvMyData, 3, dCoeff);
    printf("3rd order polynomial fit:\n");
    printf("Coeff a = %f\n", dCoeff[0]);
    printf("Coeff b1 = %f\n", dCoeff[1]);
    printf("Coeff b2 = %f\n", dCoeff[2]);
    printf("Coeff b3 = %f\n", dCoeff[3]);
}
```

The next code example is a modification of the previous example. In this example the middle third of the data set is fit.
void PolyFit2()
{
    Curve crvMyData("Data1_A", "Data1_B");
    double dCoeff[4];
    // Fit just first middle 1/3rd of dataset
    bool bRet = fitpoly(crvMyData, 3, dCoeff, 2, 3);
    printf("3rd order polynomial fit of partial data\n");
    printf("Coeff a = %f\n", dCoeff[0]);
    printf("Coeff b1 = %f\n", dCoeff[1]);
    printf("Coeff b2 = %f\n", dCoeff[2]);
    printf("Coeff b3 = %f\n", dCoeff[3]);
}

Calling NAG Functions

Origin 7 includes a number of the NAG function libraries. These libraries include:
a02 - Complex Arithmetic
c06 - Fourier Transforms
e01 - Interpolation
e02 - Curve and Surface Fitting
F - Linear Algebra
f06 - Linear Algebra Support Functions
g01 - Simple Calculations on Statistical Data
g02 - Correlation and Regression Analysis
g03 - Multivariate Methods
g04 - Analysis of Variance
g08 - Nonparametric Statistics
g11 - Contingency Table Analysis
g12 - Survival Analysis
s - Approximations of Special Functions

Before calling a NAG function from your Origin C function, you must include the header file containing the NAG function prototype. For example, for the nag_fft_complex function, you would include the following preprocessor directive before your function header:
#include <NAG\OCN_c06.h>

Alternatively, you can include OC_nag.h, which contains #include directives for all the NAG header files, but your function will take much longer to compile. To include OC_nag.h, add the following preprocessor directive to your source file:
#include <OC_nag.h>

(OC_nag.h is located in the default include folder, \OriginC\System.)

When you call a NAG function, you cannot directly pass a data set name. For example, you cannot pass a NAG function "Data1_B". Furthermore, you cannot pass a NAG function a Dataset object. Instead, your function must create a Dataset object with an attached Origin data set. You must then
create a vector object and copy the Dataset to the vector. You can then pass the vector to the NAG function.

The following example function provides a simple example of calling a NAG library function from Origin C, and then printing the results to the output window as well as to a worksheet column.

```c
#include <origin.h>
#include <NAG\OCN_g01.h>

void NAGSummaryStats(String strWksName)
{
    double dMean, dSD, dSkew, dKurt, dMin, dMax, dSum;
    int iDataLen, iWtLen, iOK;

    Dataset dsRawData(strWksName,0);
    Dataset dsWeights(strWksName,1);
    Dataset dsResults(strWksName,3);

    iDataLen = dsRawData.GetSize();

    // if dataset has less than 2 elements, no need to do statistics
    if(iDataLen < 2)
    {
        printf("Improper dataset!\r\n");
        return;
    }

    iWtLen = dsWeights.GetSize();

    // if dataset and weight columns are of unequal length, return
    if(iWtLen != iDataLen)
    {
        printf("Data and Weight columns do not match!\r\n");
        return;
    }

    // now proceed with NAG call

    // create vectors and copy dataset to these vectors in order to pass to the NAG function
    vector vV, vW;
    vV = dsRawData;
```
vW = dsWeights;

int nErr = nag_summary_stats_1var(iDataLen, vV, vW, &iOK, &dMean, &dSD, &dSkew, &dKurt, &dMin, &dMax, &dSum);
if(nErr != 0)
{
    printf("Operation Failed: NAG function error code = %d\n",nErr);
    return;
}
if(iOK != iDataLen)
{
    printf("Missing values present in the dataset!\n");
}

// printf("Operation successful \n",32);
printf("Dataset length = %d\n", iOK);
printf(" mean = %f\n", dMean);
printf(" std. dev. = %f\n", dSD);
printf(" skewness = %f\n", dSkew);
printf(" kurtosis = %f\n", dKurt);
printf(" min = %f\n", dMin);
printf(" max = %f\n", dMax);
printf("sum of weights = %f\n", dSum);

// also write results to worksheet
dsResults.SetSize(8);
dsResults[0]=iOK;
dsResults[1]=dSum;
dsResults[2]=dMean;
dsResults[3]=dSD;
dsResults[4]=dSkew;
dsResults[5]=dKurt;
dsResults[6]=dMin;
dsResults[7]=dMax;

return;
}
Statistics

Origin C provides a number of global statistics functions. These functions are prototyped in data.h.

Minimum and maximum functions are provided that accept scalars and return a scalar. For example, a max function is prototyped as follows:

```c
double max(double da, double db);
```

This function is defined in internal.c (located in the \OriginC\System subfolder) as follows:

```c
// returns max of two doubles
double max(double da, double db)
{
    return (da > db ? da : db);
}
```

Minimum and maximum functions are also provided that accept a data set and return a double. For example, a min function is prototyped as follows:

```c
double min(Dataset &aa);
```

This function is defined in internal.c as follows:

```c
// returns min value of data set
double min(Dataset &aa)
{
    BasicStats stat;
    stat.min = -1;
    Data_sum(&aa, &stat);
    return stat.min;
}
```

The following code example illustrates how you can call this min function. In this example, a worksheet name and column number are passed to the test function. The function declares a Dataset object and assigns it the passed data set. The min function is called and the result is output to the current output window.

```c
void test(string strData, int colNum)
{
    double myMin;
    Dataset dd1(strData, colNum);
    myMin = min(dd1);
    printf("Minimum: \%.1f\n", myMin);
}
```

You could then call this function from Origin using the following LabTalk command. In this call you are passing the active window name (assuming the desired worksheet is active) and the number for the second column.

```labtalk
test %H 1;
```
To compute basic summary (descriptive) statistics on an Origin data set, a `Data_sum` global Origin C function is provided. This function is prototyped in data.h as follows:

```c
BOOL Data_sum(Dataset* pData, BasicStats* pStat, Dataset* pSumData=NULL);
```

Note that in this function, `pStat` is a pointer to the returned `BasicStats` structure. After calling the `Data_sum` function, you can access the component results (such as the standard deviation) by using the `variable.member` notation, where `variable` is the variable of type `BasicStats`. This structure is defined in OC_types.h as follows.

```c
typedef struct tagBasicStats
{
    double min;
    double max;
    double mean;
    double sd;
    double total; // cumulative sum of all values
    double ssdm; // sum of squares of diff from mean
    int N; // number of good values in data set
    int Missing; // number missing vals. in data set
    int iMax; // index (0 offset) of the max value
    int iMin; // index of the min value
} BasicStats;
```

The following code example illustrates the use of the `Data_sum` function and the `BasicStats` structure.

```c
void myBasicStats(string strData, int colNum)
{
    double myTotal;
    Dataset dd1(strData, colNum);
    BasicStats mystat;
    Data_sum(&dd1, &mystat);
    myTotal = mystat.total;
    printf("Total: %.1f\n", myTotal);
}
```

You could then call this function using the following LabTalk command. In this call you are passing the active window name (assuming the desired worksheet is active) and the number for the second column.

```
myBasicStats %H 1;
```

Origin C also provides a `Data_percentiles` function for calculating percentiles at given percent values for the given data set. This function is prototyped in data.h as follows:

```c
BOOL Data_percentiles(Dataset* pData, const double v_percents[], double v_percentiles[], int nValues, BOOL bInterpolate=FALSE);
```
The following function illustrates computing percentiles using the `Data_percentiles` function. The flag value determines whether or not interpolation is performed on the data during the computation.

```c
void ComputePerc(String Wksname)
{
    Dataset col1(Wksname,0); // data column
    Dataset col2(Wksname,1); // flag column
    Dataset col3(Wksname,2); // percentile column
    Dataset col4(Wksname,3); // results column
    vector perc(), result();
    bool interp;
    // get size of perc col:
    int nperc = col3.GetSize();
    // and define vectors of that size:
    perc.SetSize(nperc);
    result.SetSize(nperc);
    // fill perc with percentiles to be computed:
    perc = col3;
    if( col2[0] == 0 )
    {
        interp = false;
    }
    else interp = true;
    // make copy of data set to pass to function:
    Dataset dd(col1);
    Data_percentiles(&dd, perc, result, nperc, interp);
    // make sure results col is of correct size:
    col4.SetSize(nperc);
    col4 = result; // copy results to col d
}
```

**Batch Processing and Automation**

Origin C provides collections for efficient looping through objects of the same type using `foreach`. For example, the `Project` class provides collections for all the windows in a project, as well as for all the worksheet, graph, notes, matrix, and layout windows in the project. Other classes also provide collections. For example, the `Page` class provides a collection of all the layers on the page.

The following example illustrates the use of the `foreach` statement and the collection of columns in a worksheet. In this example, `wks.Columns` is the collection of columns in the `wks` worksheet.
void setMyCols()
{
    // declare Worksheet object
    Worksheet wks("data1");
    foreach(Column cols in wks.Columns)
    {
        cols.SetDigitMode(DIGITS_SIGNIFICANT);
        cols.SetDigits(10);
    }
}

The next example illustrates the use of the foreach statement and the collection of pages (windows) in the project.
Project prj;
PageBase pg;
Collection<PageBase> pgcoll;
// initialize collection object from Pages property of Project class:
pgcoll = prj.Pages;
// loop over all the pages in the collection:
foreach (pg in pgcoll)
{
    string strName = pg.GetName();
    out_str(strName);
}

The next example illustrates the use of the foreach statement and the collection of data plots in a graph layer.
void setPlotColor()
{
    Project prj;
    Layer lay;
    lay = prj.ActiveLayer();
    GraphLayer glay;
    glay = (GraphLayer)lay; // explicit cast
    foreach(DataPlot dp in glay.DataPlots)
    {
        dp.SetColor(3, TRUE);
    }
}

Origin C also provides a Folder class to access Project Explorer. The Folder class includes a collection of all the subfolders in a folder. It also includes a collection of all the pages in a folder. You
can thus use a foreach statement to perform an action on all the subfolders in a folder, as well as on all the pages in a folder.

The following code creates a Folder object and assigns it the current Project Explorer folder. It then uses a foreach statement to loop through all the subfolders and output the subfolder names to the current output window. It then loops through all the windows in the current folder and outputs their names.

```c
void test()
{
    Project prj();
    String strName;
    Folder fld = prj.CurrentFolder;
    foreach(Folder sub in fld.Subfolders) {
        strName = sub.GetName();
        printf("<Folder> %s\n", strName);
    }
    foreach(PageBase page in fld.Pages) {
        strName = page.GetName();
        printf(" %s\n", strName);
    }
}
```

In addition to looping through objects in a collection with a foreach statement, you can develop code that automates multiple Origin operations. The following example function creates a new subfolder in Project Explorer with same name as the data file that was dropped. The function opens the file and reads in data to a worksheet created using a custom template. It then creates a graph of the data using a custom template, and performs a nonlinear fit of the data.

```c
void Automation(string strFileName)
{
    // Extract just the name of the file passed, from the full file name with path
    int iLen = strFileName.GetLength();
    int iSlash = strFileName.ReverseFind('\');
    int iDot = strFileName.ReverseFind('.');
    string strSampleName = strFileName.Mid(iSlash + 1, iDot - iSlash - 1);
    printf("Processing file: %s .........", strSampleName);

    // Declare root folder of PE for the current project
    Folder fldRootFolder = Project.RootFolder;

    // Create subfolder with same name as file
```
Folder fldSubFolder = fldRootFolder.AddSubfolder(strSampleName);

// Make this subfolder active
fldSubFolder.MakeActive();

// Get file path of current project
string strProjectPath;
strProjectPath = Project.GetPath();

// Create a worksheet using custom template
Worksheet wksData;
string strWksTemplate = strProjectPath + "Automation.otw";
int nOptionW = CREATE_VISIBLESAME;
bool bRetW = wksData.Create(strWksTemplate, nOptionW);

// Declare data sets in worksheet to copy data from file
Dataset dsX(wksData, 0);
Dataset dsY(wksData, 1);

// Declare a curve object using x,y columns of worksheet
Curve crvData(wksData, 0, 1);

// Get name of curve
string strYDataName;
crvData.GetName(strYDataName);

// Open file and read data into worksheet
stdioFile ffDataFile;
ffDataFile.Open(strFileName, file::modeRead);
string strSampleID, strDateTime, strTemp;
// Read three header lines
ffDataFile.ReadString(strSampleID);
ffDataFile.ReadString(strDateTime);
ffDataFile.ReadString(strTemp);
// Get number of data points, x start, and x step from header line
int iXlen = atoi(str1);
double dXstart = atof(str2);
double dXstep = atof(str3);

// Set dataset length appropriately
dsX.SetSize(iXlen);
dsY.SetSize(iXlen);

// Loop thru and read all data points - generate x and read in y from file
for(int ii = 0; ii < iXlen; ii++)
{
    dsX[ii] = dXstart + ii * dXstep;
    ffDataFile.ReadString(strTemp);
    dsY[ii] = atof(strTemp);
}

// Set Sample ID and Date/Time text labels in worksheet
// First declare graphic objects for text labels on worksheet
GraphObject grWTxtSampleID, grWTxtDateTime;
grWTxtSampleID = wksData.GraphObjects("txtID");
grWTxtDateTime = wksData.GraphObjects("txtDateTime");

// Now set text for these objects
grWTxtSampleID.Text = "\u(Sample ID:) " + strSampleID;
grWTxtDateTime.Text = "\u(Date/Time:) " + strDateTime;

// Create a graph using custom template
GraphPage grphData;
string strGrphTemplate = strProjectPath + "Automation.otp";
int nOptionG = CREATE_VISIBLE_SAME;
bool bRetG = grphData.Create(strGrphTemplate, nOptionG);

// Declare active layer in current graph page
GraphLayer grphLayer = grphData.Layers();

// Plot data curve to active layer
int nPlot = grphLayer.AddPlot(crvData, IDM_PLOT_SCATTER);
grphLayer.Rescale();

// Set Sample ID and Date/Time text labels in graph
// First declare graphic objects for text labels on graph layer
GraphObject grGTxtSampleID, grGTxtDateTime;
grGTxtSampleID = grphLayer.GraphObjects("txtID");
grGTxtDateTime = grphLayer.GraphObjects("txtDateTime");

// Now set text for these objects
grGTxtSampleID.Text = "\u(Sample ID:) " + strSampleID;
grGTxtDateTime.Text = "\u(Date/Time:) " + strDateTime;

// Perform nonlinear fit to plotted data
NLSFCntrl nlsfMyFit; // create NLSFCntrl - see OC_Types.h for details
initNLSF(nlsfMyFit); // initialize structure with default initial values
nlsfMyFit.pasteToPlot = false; // turn off pasting results to graph after fitting
lstrcpy(nlsfMyFit.szYdataName, strYDataName); // assign fit dataset name
lstrcpy(nlsfMyFit.szFuncName, "gaussamp"); // assign fitting function name
fitNLSF(nlsfMyFit); // perform fit

// Write fit parameters to graph text labels
string strCentroid, strHeight, strWidth;
strCentroid.Format("%f", nlsfMyFit.Par[1]); // Par 1, 2, 3 hold xc, w, A from fit
strHeight.Format("%f", nlsfMyFit.Par[3]);
strWidth.Format("%f", nlsfMyFit.Par[2]);

// First declare graphic objects for text labels on graph layer
GraphObject grGTxtCentroid, grGTxtHeight, grGTxtWidth;
grGTxtCentroid = grphLayer.GraphObjects("txtCentroid");
grGTxtHeight = grphLayer.GraphObjects("txtHeight");
grGTxtWidth = grphLayer.GraphObjects("txtWidth");

// Now set text for these objects
grGTxtCentroid.Text = "\u(Centroid:) " + strCentroid;
grGTxtHeight.Text = "\u(Height:) " + strHeight;
grGTxtWidth.Text = "\u(Width:) " + strWidth;

// Close data file
ffDataFile.Close();
// Set PE folder path back to root folder, for processing next file if any
fldRootFolder.MakeActive();
printf("done!\n");
}

### Calling LabTalk and Using LabTalk Variables

There are a number of strategies for including LabTalk code in your programs:

1) Use the `LT_get_var` and `LT_get_str` global functions to get LabTalk variable values.
2) Use the `LT_set_var` and `LT_set_str` global functions to set LabTalk variable values.
3) Use the `LT_evaluate` global function to evaluate a LabTalk expression.
4) Use the `LT_execute` global function to execute a LabTalk statement.
5) Exclude a part of your source code from compiling by using the `#ifdef` and the `#endif` preprocessor directives. Include LabTalk code between these directives.
6) Include an `_LT_obj { }` block in your function to code LabTalk object property and method statements.

### Getting and Setting LabTalk Variable Values

Origin C provides global functions to get and set the value of LabTalk numeric and string variables. These variables include user-defined LabTalk variables, LabTalk system variables, and LabTalk object properties. LabTalk system variables are variables that are used internally by Origin. LabTalk object properties are attributes of LabTalk objects. Their syntax is `objectName.property` for numeric values and `objectName.property$` for string values.

*The LabTalk numeric system variables include (but are not limited to):*

- **X1, X2, Y1, Y2, Z1, Z2**: When a graph window is active, these variables contain the initial and final X, Y, and Z axis scale values for the active layer. Changing these values resets the X, Y, or Z scale.
- **X, Y, Z**: These variables contain the current X, Y, and Z coordinates of the Screen Reader, Data Reader, or Data Selector tools.
- **i**: The variable i is used to store the index number to a data set. In most cases, this is also the row number of a worksheet. (Notice that in LabTalk, the data set index starts at 1. This contrasts with Origin C, in which the index starts at 0).
- **SELC1, SELC2, SELR1, SELR2**: These variables contain the column and row numbers that delineate the current worksheet selection range.
- **@A**: The angular unit setting. 0 = radian, 1 = degree, 2 = gradian.
- **@D**: The current date and time in Julian days.
@SD: Control the number of significant digits displayed when mathematical operations are performed in the Script window.

*The LabTalk string system variables include (but are not limited to):*

%-C: The name of the current active data set.
%-G: The current project name.
%-H: The current active window title.
%-X: The DRIVE:PATH of the current project.
%-Y: The DRIVE:PATH for the ORIGIN70.INI file.

*Getting and setting numeric variable values:*

To **get** the value of a LabTalk numeric variable from Origin C, use the following global function:

```c
BOOL   LT_get_var(LPCSTR lpszLabTalkVarName, double * lpdbResult);
```

where:

- `lpszLabTalkVarName` = LabTalk numeric variable name (user-defined variable, system variable, or object property). Must be surrounded in quotation marks. Case insensitive.
- `lpdbResult` = result of getting LabTalk variable value.

This function returns `TRUE` if `lpszLabTalkVarName` is a LabTalk variable. Otherwise, this function returns `FALSE`.

To **set** the value of a LabTalk numeric variable from Origin C, use the following global function:

```c
BOOL   LT_set_var(LPCSTR lpszLabTalkVarName, double dbValue);
```

where:

- `lpszLabTalkVarName` = LabTalk numeric variable name (user-defined variable, system variable, or object property). Must be surrounded in quotation marks. Case insensitive.
- `dbValue` = value to set the LabTalk variable to.

This function returns `TRUE` if `lpszLabTalkVarName` is a LabTalk variable. Otherwise, this function returns `FALSE`.

The following example checks if the axes are displayed in the layer. It outputs 0 if not displayed and 1 if displayed.

```c
void checkValue( )
{
    double vv;
    LT_get_var("LAYER.SHOWX", &vv);
    printf("The value is: %.1f\r\n", vv);
}
```

The next example gets and sets the X coordinate of the Screen Reader, Data Reader, or Data Selector tool.
double vv;
if (LT_get_var("X", &vv))
{
    vv += 10;
    if (LT_set_var("X", vv))
        out_str("set X in LabTalk succeeded");
    else
        out_str("set X in LabTalk failed");
}

Getting and setting string variable values:
To get the value of a LabTalk string variable from Origin C, use the following global function:

BOOL   LT_get_str(LPCSTR lpcszLabTalkString, char* pBuffer, int nBufferSize);

where:

lpcszLabTalkString = LabTalk string variable (%A - %Z) or LabTalk object string property.
Must be surrounded in quotation marks. Case insensitive.
pBuffer = character buffer to receive the string.
nBufferSize = size of the buffer including the terminating zero.
This function returns TRUE if lpcszLabTalkString is a LabTalk string variable. Otherwise, this function returns FALSE.

To set the value of a LabTalk string variable from Origin C, use the following global function:

BOOL   LT_set_str(LPCSTR lpszVar, LPCSTR lpszContent);

where:

lpszVar = LabTalk string variable (%A - %Z) or LabTalk object string property. Must be surrounded in quotation marks. Case insensitive.
lpszContent = string content to be copied to the string variable. Can be NULL to empty a string.

The following example gets and outputs the name of the active data set.

void getName()
{
    char szTemp[100];
    LT_get_str("%C", szTemp, 100);
    string strA = szTemp;
    printf("The active data set is: %s\n", strA);
}
Evaluating LabTalk Expressions

Origin C provides the following global function to evaluate a LabTalk expression using the LabTalk interpreter and assign the result to an Origin C variable:

```c
BOOL LT_evaluate(LPCSTR lpszLabTalkExpression, double * lpdbResult);
```

where:

- `lpszLabTalkExpression` = LabTalk expression. Must be surrounded in quotation marks. Case insensitive.
- `lpdbResult` = result of the expression evaluation.

This function returns `TRUE` if `lpszLabTalkExpression` is a LabTalk expression that can be evaluated by the LabTalk interpreter. Otherwise, this function returns `FALSE`.

Note: This function is similar to the `LT_get_var` function in that it can also get the value of a LabTalk variable. However, using the `LT_evaluate` function to get the value of a variable is slower than using the `LT_get_var` function. Furthermore, the `LT_get_var` function will recognize that a given name corresponds to a LabTalk variable, even when the variable's value is a missing value. This contrasts with the `LT_evaluate` function, which will return `FALSE` (0) because it will not recognize the LabTalk variable.

In the following example, an expression is evaluated by the LabTalk interpreter and assigned to the `vv` variable.

```c
double vv;
LT_evaluate("0^0", &vv);
printf("vv = %f\n", vv);
```

Executing a LabTalk Statement

Origin C provides the following global function to execute a LabTalk statement:

```c
BOOL LT_execute(LPCSTR lpszLabTalkStr, int wCntrl = 0);
```

where:

- `lpszLabTalkStr` = LabTalk statement to execute, without the terminating semicolon. Must be surrounded in quotation marks. Case insensitive.
- `wCntrl` = for future implementation only. Default value is zero so omit second argument or set as zero.

This function returns `TRUE` if `lpszLabTalkStr` is a LabTalk statement that can be executed. Otherwise, this function returns `FALSE`. 
The following example executes a LabTalk statement and gets and outputs the result.

```c
void test()
{
    LT_execute("x=sqrt(100) ");
    double vv;
    LT_get_var("x", &vv);
    out_double("sqrt(100) = ", vv);
}
```

**Including LabTalk Code in Your Origin C Source File and Blocking it from Compiling**

By combining a few programming techniques, you can include LabTalk code right in your Origin C source file, exclude the LabTalk code from compiling, and then call that code from your Origin C function. This strategy takes advantage of the following programming techniques:

1) The Origin C compiler supports preprocessor directives to exclude part of your source code from compiling. In this case, the LabTalk section of code will be excluded.

2) The Origin C global function `LT_execute` lets you execute a LabTalk statement from your Origin C function.

3) LabTalk provides a `run.section` method to execute a section of LabTalk script. The section of LabTalk script must begin with a section name surrounded by brackets. For example:

```
[MySectionName]
```

The following example illustrates this programming strategy. Assume that this code is in a source file named `myFile.c`.

```c
void myfunc()
{
    LT_execute("run.section(myFile.c, Main) ");
}
```

```c
#ifdef LABTALK // can be anything not defined
  [Main]
      LabTalk code goes here
#endif
```

When you compile this source file, the Origin C preprocessor runs. The preprocessor first encounters the `#ifdef LABTALK` directive. Because `LABTALK` is not defined using a `#define` directive, the preprocessor will begin excluding code from the compiler until it reaches the `#endif` directive. Thus, all the LabTalk code is excluded from the compiler.

The output from the preprocessor is then sent to the Origin C compiler, which creates the machine code in a separate file. Keep in mind that the source file, `myFile.c`, still exists.

When you call the Origin C `myFunc` function from LabTalk, the Origin C built-in `LT_execute` function is called. This function executes a LabTalk statement. In this case, the LabTalk statement is
a `run.section` method, which runs the LabTalk script in the `myFile.c` file located in the section named `[Main]`. Of course `myFile.c` is your source file! Your LabTalk code in this section is then executed.

**Executing a Block of LabTalk Object Code**

You can include LabTalk script containing object property and method statements in your Origin C functions using the following syntax:

```c
_LT_obj
{
    // this block allows LabTalk objects
    // and associated syntax, for example:
    stat.data$ = nlsf.x$;
}
```

Because this block allows both LabTalk and Origin C, all expressions are first assumed to be Origin C. If the expression is not a valid Origin C expression, then it is assumed to be LabTalk and is sent to the LabTalk interpreter for evaluation.

This LabTalk object support is particularly useful for accessing LabTalk objects that do not have an Origin C counterpart, or that have limited implementation in Origin C.

The following list clarifies the LabTalk and Origin C support in the `_LT_obj` block:
1) LabTalk variable assignments *cannot* be made in this block. For example, `ii = 5;` is illegal and will generate a compile error.
2) Origin C variables must be declared *before* this block (if they are accessed in the block).
3) Arguments in LabTalk object methods must be Origin C variables or expressions.
4) The return value of LabTalk object methods must be assigned to Origin C variables.
5) You *cannot* use LabTalk commands in this block - only LabTalk object notation.

**Calling Origin C Functions from Origin**

Once your Origin C function is loaded in the current Code Builder workspace and compiled and linked, you can access that function in Origin by executing a LabTalk statement that calls your function. The general syntax for this function call is:

```c
funcName(arguments);
```

or if the Origin C function returns a value, then the syntax is:

```c
varName = funcName(arguments);
```

When "`funcName(arguments);`" is passed to the LabTalk interpreter, the interpreter will check if `funcName` is a compiled Origin C function. If it is, the Origin C function will be executed.
For example, if your Origin C function header is:
void doc(string str, int nn)
You could call this compiled function using the following LabTalk statement:
doc(%H, 3);

In addition to using this general syntax, you can also call your function using the following LabTalk syntax (omitting the parentheses):
funcName arguments;
For example, you could call the doc function using the following LabTalk statement:
doc %H 3;

**Passing Arguments to Your Origin C Function**

When you call your compiled Origin C function from LabTalk, the Origin C function will also be checked. The Origin C function can only take numeric and string arguments. Thus, only the following argument types are supported: char, short, WORD, int, DWORD, float, double, and string. In the case of arguments of type double, Origin C will also accept vector notation (see item 2).

1) If the Origin C function's formal parameter is an int, then the LabTalk variable that is passed as an argument to this function will be converted to an int.

For example, if you compile the following Origin C function:
void testNumeric(int vv)
{  
   printf("The number is %d", vv);
}
Then the following LabTalk function call:
testNumeric(5.6);
Returns:
The number is 6

2) If the Origin C function's formal parameter is type double and the function returns a double, then the function also supports vector notation. This means that you can pass a data set to the function and the function will return a vector which you can assign to a data set. For example, if you compile the following Origin C function:
double test(double var1, double var2)
{
   return var1 + var2;
}
You can then call this function from LabTalk as follows:
col(1) = 3 * test(col(2), col(3));
3) If the LabTalk variable that is passed as an argument to the Origin C function is a string, how Origin C handles the string depends on whether or not the string is surrounded by quotation marks in the function call. If the string is surrounded by quotation marks, then it is considered to be an Origin C string literal (constant). Otherwise, it is considered to be a string variable or an object string property. In this case, the string variable or object string property is first evaluated by the LabTalk interpreter and replaced with the string value that it represents.

For example, if you compile the following Origin C function:

```c
void testString(string str)
{
    printf("The string is \"%s\"\n", str);
}
```

Then the following LabTalk function call:

```labtalk
testString(%c);
```

Returns:

```
The string is "Data1_B"
```

Assuming Data1_B is the active data set.

Whereas the following LabTalk function call:

```labtalk
testString("%c");
```

Returns:

```
The string is "%c"
```

----

In addition to LabTalk string variables, you can also pass LabTalk string object properties to Origin C functions. For example, the following LabTalk function call:

```labtalk
testString(nlsf.y$);
```

Returns:

```
The string is "Data1_B"
```

Assuming Data1_B is the data set assigned to this object property.

### Returning a string to the LabTalk Calling Function

LabTalk supports Origin C functions that return `void`, `numeric`, `vector`, or `string`. If the Origin C function has a return value of type `string`, your LabTalk call to this function must always terminate with the dollar sign character ($). For example, the following function returns an error message from an error code parameter.
string getErrorMessage(int iErrorCode)
{
    string strVarType;
    switch( iErrorCode )
    {
        case 111:
            strVarType = "Error message passed for code 111";
            break;
        case 222:
            strVarType = "Error message passed for code 222";
            break;
        case 333:
            strVarType = "Error message passed for code 333";
            break;
        default:
            strVarType = "--";
    }
    return strVarType;
}

You can call this function from LabTalk as follows:
%A = getErrorMessage(333)$;

---

**Returning a vector to the LabTalk Calling Function**

You can define an Origin C function that returns a vector to the LabTalk calling function. In this case your LabTalk call to your Origin C function can assign the Origin C function's return value to an Origin data set. Thus, if your Origin C function header is:

```
vector   myFunc(
    parameterList
)
```

Then the following LabTalk call to this compiled function will assign the return value (a vector) to the data1_b data set:

```
data1_b = myFunc(argumentList);
```

Vector notation can also be used if your Origin C function's formal parameter is type double and the function's return type is double. In this case, you can pass a data set to the function and the function will return a vector which you can assign to a data set. This vector notation is illustrated in the following example.

```
double square(double var)
{
    return var * 2;
}
```
You can call this function from LabTalk as follows:

\[ \text{col}(2) = \text{square}(\text{col}(1)) \]

### A Note About Calling Global Origin C Functions from LabTalk

Like user-defined Origin C functions, you can call global Origin C functions from LabTalk if the following conditions are met:

1. The function has been defined and compiled. Thus, global Origin C functions that are prototyped - but have no function body - cannot be called. NAG functions fall into this category and thus cannot be called from LabTalk (only from Origin C).
2. All arguments must be numeric or string.
3. The return type can be numeric, string, or vector.

### Listing the Origin C Functions that Can be Called From LabTalk

To list the Origin C functions that can be called from LabTalk, use the LabTalk `list` command as follows:

```labtalk
list f;  // list all Origin C functions that can be called from LabTalk
list fs;  // only those returning string
list fv;  // only those returning vector
list fn;  // only those returning numeric
list fo;  // only those returning void
```

To show the function prototype in the listing, use `'cf'` in place of `'f'`. For example, use:

```labtalk
list cf;
lis cfs;
```

etc.

If you want to restrict LabTalk function calls to only LabTalk functions (excluding Origin C functions), you can modify the value of the `@OC` LabTalk system variable.

- `@OC = 1` (default), you can call Origin C functions from LabTalk
- `@OC = 0`, you cannot call Origin C functions from LabTalk

The `@OC` system variable setting effects the function listing with the `list` command.
Precedence Rules When a LabTalk and Origin C Function Have the Same Name

When you call a user-defined or global Origin C function from LabTalk, if the Origin C function name is the same as a LabTalk built-in function, then the Origin C function has higher precedence, except when using LabTalk vector notation. Thus, LabTalk functions like `Normal` and `Data` (which return a range of values and are thus used in vector notation) would be higher precedence than Origin C functions of the same name. In all other cases, the Origin C function is called.

Creating a GUI to call Your Function from Origin

After you have compiled your Origin C function(s), you'll need to call your function from Origin. You could call your function from the Script window. However, if you call your function often you may want to develop a more permanent calling method.

There are many options for initiating your program from Origin, including (but not limited to):

1) Clicking the Custom Routine button on the Standard toolbar.
2) Clicking a new toolbar button that you create.
3) Clicking a button located on an Origin window (a programmed object).
4) Selecting a menu command that you create.

Note: OriginPro provides additional options for creating custom applications, including using Microsoft Visual C++ resources to build floating tools, dialog boxes, and wizards. Origin's NLSF wizard and many of the floating tools (such as the One-way and Two-way ANOVA tools) were created using OriginPro.

The Custom Routine Button

The Custom Routine button on Origin's Standard toolbar executes the LabTalk script located in the [Main] section of the Custom.ogs file which is located in your Origin folder. By default, the LabTalk script in the [Main] section is a `type` statement, which displays a message in an Attention dialog box. To initiate your custom program from this button, you must replace this `type` statement with a LabTalk call to your function.

To open the Custom.ogs file in Code Builder, press CTRL+SHIFT and click the Custom Routine button on Origin's Standard toolbar. The script file opens in Code Builder. (Alternatively, click the Open button in Code Builder and then select 'LabTalk Script File (*.OGS)' from the Files of Type drop-down list and browse to and open Custom.ogs.)
Once the script file is open in Code Builder, replace the following LabTalk statement:

```labtalk
type -b $General.Userbutton;
```

with statements to compile, link, and call your source file. After making these changes, save the Custom.ogs file. You can now initiate your custom program by clicking the Custom Routine button in Origin.

For example, if you program the following function in the MyFile.c source file located in the /OriginC subfolder:

```c
void test()
{
    printf("Hello World\n");
}
```

Then you could enter the following LabTalk code in the [Main] section to compile, link, and call your function:

```labtalk
if (run.LoadOC("MyFile")!=0)
    type -b "Cannot load file";
else
    test;
```

Save the script file after adding your code.

### New Toolbar Buttons

In addition to using the Custom Routine button on Origin's Standard toolbar to initiate your program, you can create your own button and add it to an existing or new toolbar. Though the Custom Routine button provides an advantage because it is already "hooked up" to a LabTalk script file, you can create a new script file and associate it with your new button.

Script files are ASCII files that contain LabTalk script. Origin uses the .OGS extension as a way of easily recognizing script files - but they can have any extension. Script files can (and should) contain sections, so that your script can be made modular. Section names are surrounded by brackets [ ]. For example, a script file could contain the following sections:

```labtalk
[SectionA]
    include LabTalk statements here
[SectionB]
    include LabTalk statements here
[SectionC]
    include LabTalk statements here
```

If your script file is only making a call to your Origin C function, you would only need one section in your script file for this function call.
Creating a Script File that Calls Your Origin C Function

To create a new script file, open Code Builder and then select File:New from the Code Builder menu. This command opens the New File dialog box. Select LabTalk Script File from the upper list box and then type a name in the File Name text box. Specify a location and then click OK to create the script file. The blank script file displays in the Code Builder multiple document interface (MDI).

Your new script file will need a section name. For example, to add a section named Main, type the following in your script file and press ENTER:

```
[Main]
```

Next, you'll need to include code to compile, link, and call your source file.

For example, if you program the following function in the MyFile.c source file located in the /OriginC subfolder:

```c
void test()
{
    printf("Hello World\n");
}
```

Then you could enter the following LabTalk code in the [Main] section to compile, link, and call your function:

```labtalk
if (run.LoadOC("MyFile")!=0)
    type -b "Cannot load file";
else
    test;
```

Save the script file after adding your code.

Creating the Toolbar Button to Run Your Script File

In addition to creating your script file to call your Origin C function, you need to create your new toolbar button to execute the LabTalk script in your script file. To do this, select View:Toolbars (in Origin) to open the Customize Toolbar dialog box. If you want your button located on a new toolbar, click the New button on the Toolbars tab and then type the desired name in the New Toolbar dialog box. If you want your button located on an existing toolbar (or after having created your new toolbar), select the Button Groups tab.

The Groups list box displays all the button groups (toolbars) that are currently available. When a group is selected, the associated buttons for the group display in the Buttons area. The User Defined button group is located at the bottom of the Groups list box. This group contains 10 buttons that are not pre-programmed to perform a task. You can thus associate your script file with one of these buttons.

To program one of these buttons, click on the desired button in the Buttons area and then click on the Settings button. The Button Settings dialog box opens. Specify the name of the LabTalk script file containing your script (function call), the section name of the script file, any arguments to pass to the function, and other button properties. After editing this dialog box and clicking OK, drag the button that you just programmed onto your new toolbar or onto an existing toolbar. The button is ready to use.
Objects on Windows

In addition to programming a toolbar button to initiate your custom program, you can program an object on a window to initiate your program. All objects such as text labels, rectangles, lines, arrows, and other annotations have associated Label Control dialog boxes. The Label Control dialog box is used to name the object (for programmatic control) and to define the object's script and script execution trigger. An example of a script execution trigger is to display the object as a button that can be clicked to run the associated script.

Creating the Object

The annotation objects are available from the Tools toolbar. You can locate an object that you want to program directly on a worksheet, graph, or layout page window. For graph windows, if you do not want the object included when printing, copying, or exporting the graph, then you should locate the object off the page (in the gray area).

Programming the Object

To associated LabTalk script with the object, press ALT while double-clicking on the object. Alternatively, select the object and then select Format:Label Control. Your code to compile, link, and call your Origin C function is added to the lower text box.

For example, if you program the following function in the MyFile.c source file located in the /OriginC subfolder:

```c
void test()
{
    printf("Hello World\n");
}
```

Then you could enter the following LabTalk code in the lower text box to compile, link, and call your function:

```labtalk
if (run.LoadOC("MyFile")!=0)
    type -b "Cannot load file";
else
    test;
```

After typing your code in the lower text box, select the script execution method from the Script, Run After drop-down list. For example, select Button Up to display the object as a button that you can click on to execute the script.

Saving the Window as a Template

Once you have created and programmed your object, you can save the window that the object is located on as a template. You can then open a new window based on that template for easy reuse.
Menu Commands

In addition to programming a toolbar button or an object on a window, you can program a new menu command to initiate your custom program. To do this, you must create a configuration file that defines the new menu command, including the script that the menu command runs. You must also update the Origin.ini file so that your configuration file is read when Origin starts.

To create a new configuration file for your menu command, open Code Builder and then select File: New from the Code Builder menu. This command opens the New File dialog box. Select Text File from the upper list box and then type a name in the File Name text box. Specify a location and then click OK to create the configuration file. The blank configuration file displays in the Code Builder MDI. Then type the LabTalk script that defines the new menu command and its associated script. Use the LabTalk menu command to accomplish this. Save the file from Code Builder when done entering your LabTalk script. (Important Note: Code Builder will save the file with a .TXT extension. You must change this extension to .CNF in Windows Explorer.)

The following configuration file example creates a View menu command that is available when a worksheet or graph is active. The command is named Output Hello World. This menu command will compile, link, and call your function (test). The function's source file (myFile.c) is located in the OriginC\myProgram subfolder.

```
menu -wg;  // available for worksheet or graph
menu 3;  // activate View menu
// define menu command text and script
menu (Output Hello World)
{
    if (run.LoadOC("myProgram\myFile")!=0)
        type -b "Cannot load file";
    else
        test;
};
```

In order for Origin to execute this script when starting, you must include the name of the file in the Origin.ini file. To do this, open the Origin.ini file in Code Builder (or any text editor). This file is located in your Origin program folder. In the [Config] section of the file, you will see the following:

```
Title1=Full Menus
File1=Macros FullMenu
; The names in this list are config file names.
; They are assumed to have the file extension .CNF.
; Config files must be ASCII files written in the LabTalk script language.
; MACROS.CNF: basic macros used by other config files
; FULLMENU.CNF: adds to the menu structure for each window
```
Edit the `File1=...` line so that it looks like the following (keep the other lines the same):

```
File1=Macros FullMenu MyConfigFile
```

where `MyConfigFile` is the name of your configuration file. Save the file with your changes. Restart Origin and you will see your `Output Hello World` menu command under the `View` menu. Select the menu command to initiate your program.

---

**Origin C's Integrated Development Environment**

The Origin C integrated development environment is called Code Builder. Code Builder provides modern tools for writing, compiling, and debugging your code.

When you add a file to the Code Builder workspace, a file icon is added to the workspace tree in the Workspace window. You can use this tree to navigate between files in your workspace. When you double-click on the file icon in the tree, the file is activated (or opened) in the multiple document interface (MDI).
Double-click to activate the file in the MDI.

You can also use the workspace tree to navigate within a file. After you compile the functions in a source file, you can display the function definitions, function declarations, classes, structures, and type definitions encountered during compilation in the workspace tree. You can then double-click on the associated icon to activate (or open) the source file in the MDI and activate the cursor at the associated line of code.
Code Builder also provides tools for locating files that contain a specified character string. The search results are output to the Output window. If occurrences of the string were found, you can double-click on the line in the Output window to open the file in the MDI with the line containing the search string marked.
Text Editor Windows

Code Builder provides a text editor window for viewing, editing, and printing source, header, LabTalk script, and text files. The text editor window supports standard keyboard and mouse text selection and cursor control. You can also perform the following editing operations:

=> Cut, copy, paste, and delete text.
=> Undo and redo your editing actions.
=> Drag selected text between text editor windows.

The text editor automatically indents your text depending on the previous lines of code. For example, if the previous line is an open brace \{, the text editor will automatically indent the next line of code.

To make your code easier to read and edit, the text editor automatically displays different elements of your code using different colors.
Code Builder provides tools for searching for strings in the active MDI file or in multiple files.
Text editor windows support standard keyboard and mouse file navigation methods. In addition, you can navigate in a file using the following techniques:

1) Using Bookmarks. Bookmarks allow you to flag lines of code so that you can easily return to the line later.

2) Using the Go To Matching Brace. When the cursor is active before or after an opening or closing brace {}, bracket [], or parenthesis (), click the Go to Matching Brace button on the Edit toolbar to highlight the matching brace.

3) Find dialog box. If you want to go to the section of your file that contains a specific string, enter your string in the search string combo box located on the Search toolbar or in the Find dialog box (click the Find button on the Edit toolbar).

4) Workspace tree. For each compiled source file, you can activate the display of workspace tree icons for the function definitions, function declarations, classes, structures, and type definitions encountered during compilation. You can then double-click on an icon in the workspace tree (for example, a function definition) to go to the respective location in the source file.

Compiling

Code Builder provides a number of methods for compiling and performing linking on files that have been added to your workspace. You can select individual or multiple files to compile, or you can build the workspace with the option of excluding specified files.
When you compile and link source files or build all non-excluded files in the workspace, the compiling and linking results are displayed to the Output window.

=> If the compiling and linking was successful, the Output window lists the source files that were compiled. The *Done!* line indicates success.

=> If errors were encountered during the compiling and linking process, the Output window lists the file name, line number, and the error encountered. You can double-click on the error line in the Output window to activate the source file and show the error line.
Debugging

The first step in the debugging process is to correct all syntax, spelling, and type errors so that your function(s) properly compile. Once you eliminate the compile-time errors, you can then test for logic errors using Code Builder's debugger.

You can test run your function(s) from Code Builder's LabTalk Console. Click in the upper pane of the LabTalk Console, type your function call, and then press ENTER.
If your program does not run as expected, you can find and correct the logic errors using Code Builder's debugger. To locate logic errors in your program, you can step through your code by pausing execution at breakpoints and checking the values of variables. Code Builder provides Step Into, Step Out Of, and Step Over breakpoint tools.
When the debugger is stopped at a breakpoint, you can view the value of a variable by placing the mouse pointer over the variable in your source file. The current variable value displays in a pop-up box. You can also use the Variables window and the Watch window to view the value of variables while debugging.

If the function that you are calling to start your program contains calls to other functions, you can use the Call Stack window to see the name of the function (and source code line number) that is currently executing as well as the functions that have been called but have not yet returned control to the calling function.

You can also use the ASSERT macro to identify logic errors during program development.
**ASSERT(condition)**

When a function is called that includes an `ASSERT`, the `ASSERT` outputs a diagnostic message to Code Builder's Output window when `condition` evaluates to false (zero). The diagnostic message indicates the source file and line number where the assertion failed.

In the following example, if the size of the vector `bb` is not equal to the size of the vector `aa`, then a diagnostic message displays in the Output window.

```c
void test(int imax = 10)
{
    vector<int> aa(imax);
    vector<int> bb;

    for (int ii = 0; ii < aa.GetSize(); ii++)
        aa[ii] = ii;

    bb = 10 * aa;

    ASSERT(bb.GetSize() == imax);
    for (ii = 0; ii < bb.GetSize(); ii++)
        printf("bb[%d]=%d \n", ii, bb[ii]);
}
```
Index

% 
%C active data set 
   LabTalk system variable 74
%G name of current project (.OPJ) 
   LabTalk system variable 74
%H active window name 
   LabTalk system variable 74
%X path of current project (.OPJ) 
   LabTalk system variable 74
%Y drive and path of Origin 
   LabTalk system variable 74

@ 
@A 
   angular units 
   LabTalk variable 73
@D 
   current date and time 
   LabTalk variable 73
@SD 
   significant digits for math 
   LabTalk variable 74

_ 
_LT_obj 73

A
Analysis 
   differentiation 60
   integrating 60
   linear and polynomial regression 
      61
   nonlinear curve fitting 58, 69
   set data set range 15
   statistics 65
Angular units

LabTalk variable 73
Annotations 
   deleting 23
ANSI C support 
   Origin C 1
Application development 
   running application from Origin 83
   writing your code 1
Arguments 
   default values
      C++ support 4
      passing by reference 
      C++ support 3
      passing to Origin C functions 
      from LabTalk 79
Arrays 7
   dynamic 1-D 36
   dynamic 1-D and 2-D 41
   dynamic 2-D 25
   support of 1
ASCII data files 
   importing 52, 69
ASSERT macro 
   debugging 97
Automation 67
Axes 
   initial and final scale 
   LabTalk variable 73
   rescaling after setting data range 
      20

B
Basic data types 6
Batch processing 67
Binary data files 
   importing 18, 32, 52
Bookmarks 
   navigating in a source file 93
BOOL 6
Breakpoints 
   debugging 96
Button Groups tab 
   Customize Toolbar dialog box 85
Buttons 
   on toolbar 
      programming 83, 84
   on window 
      programming 86
BYTE type 6

C
C See Origin C
C support
  Origin C differences 2
C# support
  collections 5
  foreach statement 5
C++ support
  class support 3
  default argument values 4
  function overloading 2
  passing arguments by reference 3
  variable declarations 2
Call stack
  debugging 97
Calling
  Origin C functions
    from LabTalk 78
  Origin C global functions
    from LabTalk 82
Categorical data 11, 12
CategoricalData class 11
CategoricalMap class 12
char type 6
Character strings 40
  string class 33
Charts
  See Graphs
Classes
  CategoricalData 11
  CategoricalMap 12
  Collection 12
  Column 13
  Complex 14
  Curve 14
  DataObject 15
  DataPlot 16
  Dataset 16
  Datasheet 17
  file 18, 52
  Folder 18
  GraphLayer 20
  GraphObject 21
  GraphPage 22
  Layer 23
  Layout 24
  LayoutPage 24
  matrix 25
  Matrix 26
  MatrixLayer 27
  MatrixObject 27
  MatrixPage 28
  Note 29
  OriginObject 29
  Page 29
  PageBase 30
  progressBox 30
  Project 31
  ROIObject 32
 stdioFile 32, 52
  string 33
  StyleHolder 34
  vector 36
  waitCursor 36
  Worksheet 37
  WorksheetPage 38
Code Builder
  compiling and linking files 93
Collection class 12
Collections
  C# support 5
  columns in worksheet 13, 15, 37, 38
  data plots in layer 20
  graphic objects in layer 21, 23
  graphic objects in layout page 24
  layers in graph 16, 20, 29, 34
  matrix objects in matrix 27
  Project Explorer subfolders 18
  Project Explorer windows 18
  style holders in layer 20, 34
  summary of class 12
  windows in project 31
Column class 13
Columns (worksheet)
  adding to worksheet 37
  counting 17
  formatting 13
  math operations 57
  operations on 37, 56
  renaming 37
  set as categorical 11
  setting numeric display 38
  setting range 15
  setting values 16, 56
Compiling Origin C functions
  Code Builder 93
Complex class 14
Complex numbers 14, 43
Configuration files
  programming menu command 87
Curve class 14
Custom Routine button
  programming 83
Customize Toolbar dialog box 85

D
Data acquisition
  GPIB 46
  RS-232 43
Data labels
add to graph 20
Data plots
  add to graph 20
  grouping and ungrouping 20
Data Reader
  coordinates
    LabTalk variable 73
Data Selector
  coordinates
    LabTalk variable 73
Data sets
  accessing elements 17
  active
    LabTalk system variable 74
  assigning values from vector
    object 81
  creating 16, 56
  importing and exporting 18, 32, 52
  index number
    LabTalk variable 73
  math operations 57
  significant digits 74
  set as categorical 11
Data types
  basic 6
  composite 7
  derived 7
DataObject class 15
DataPlot class 16
Dataset class 16
Datasheet class 17
Date
  LabTalk variable 73
Debugging
  ASSERT macro 97
  call stack 97
  tips 95
  using breakpoints 96
Declarations
  C++ support 2
Degrees
  LabTalk variable 73
Differentiation
  60
  double type 6
  DWORD type 6

stdioFile class 52
extern storage class specifier
  on functions 1

F
file class 18, 52
Files
  Code Builder
    navigating in 93
    importing and exporting data 18, 32, 52
  Fitting
    See Nonlinear curve fitting
  float type 6
  Floating types 6
  Folder class 18
foreach statement
  C# support 5
  columns in a matrix 28
  columns in worksheet 37, 38
  data plots in layer 20
  graphic objects in layer 21, 23
  graphic objects in layout page 24
  Project Explorer subfolders 18
  Project Explorer windows 18
  setting column ranges 15
Fortran routines
  calling from Origin C 48
Function overloading 2
Function pointers
  not supported 1
Functions
  calling from LabTalk 78
    passing arguments 79
  compiling and linking 93
  debug
    call stack 97
    setting breakpoints 96
  extern storage class specifier 1
  listing those you can call from LabTalk 82
  precedence rules for LabTalk and Origin C 83
  returning string to LabTalk 80
  returning vector to LabTalk 81

G
Global functions
  Origin C
    accessing LabTalk 73
    differentiation 60
    integrating data 60
    percentiles 66
statistics 65
Go to matching brace 93
navigating in a source file 93
GPIB
data acquisition 46
Gradians
LabTalk variable 73
Graphic object
accessing 21
deleting 23
GraphLayer class 20
GraphObject class 21
GraphPage class 22
Graphs
accessing graphic objects 21
accessing layers 23
adding data to layer 20
appending layers 22
categorical data 12
creating from template 22, 30
data set plotting designations 34
grouping and ungrouping data 20
numbers of layers 13, 30
plotting data 16, 20
rescaling axes 20
setting data plot range 20
style holders 34
Grouping/ungrouping data plots 20

H
Handle
importing and exporting data 18,
32, 52
Hourglass wait cursor
displaying 36

I
i
  data set index
    LabTalk variable 73
I/O
code examples 39
file class 18, 52
stdioFile class 32, 52
Importing
data
file class 18, 32, 52
stdioFile class 52
IMSL function
Fortran routines
calling from Origin C 50
Index number
  data set
    LabTalk variable 73
int type 6
Integral types 6
Integrating data 60

J
Julian days
current date and time
  LabTalk variable 73

L
Label Control dialog box 86
LabTalk
calling Origin C functions 78
  passing arguments 79
calling Origin C global functions 82
  including in Origin C 73
  listing Origin C functions 82
  Origin C returns string 80
  Origin C returns vector 81
  preventing calls to Origin C 82
  script files 84
Layer class 23
Layers
  accessing 20
  adding data 20
  counting all in graph 13, 30
Layout class 24
Layout pages
  accessing graphic objects 24
  creating 24
LayoutPage class 24
Linear regression 61
Linking
  files in Code Builder 93
long type 6
LT_evaluate 73
LT_execute 73
LT_get_str 73
LT_get_var 73
LT_set_str 73
LT_set_var 73

M
Math operations
  on data sets 57
  significant digits 74
Matrices
  accessing 26
accessing elements 26
creating from template 28
dimensions 27
flipping and rotating 27
formatting 27
number of columns and rows 27
opening 27
setting values 26
matrix class 25
Matrix class 26
MatrixLayer class 27
MatrixObject class 27
MatrixPage class 28
Menu commands
programming 87
Multi-dimensional arrays
support of 1

N
NAG
calling a function 62
supported function libraries 62
Navigating
go to matching brace 93
in a single source file 93
using bookmarks 93
NLSF See Nonlinear curve fitting
Nodes
graphic objects 21
Nonlinear curve fitting 58, 69
Note class 29
Notes windows
creating 29

O
Objects
on window
programming 86
Operators
exponentiate vs. bitwise exclusive
OR 2
Origin
drive and path location
LabTalk system variable 74
programming strategies 83
Origin C
ANSI C support 1
base class
OriginObject 29
basic data types 6
C# support 5
C++ support 2
CategoricalData class 11
CategoricalMap class 12
Collection class 12
Column class 13
Complex class 14
Curve class 14
DataObject class 15
DataPlot class 16
Dataset class 16
Datasheet class 17
derived types 7
file class 18, 52
Folder class 18
functions
calling from LabTalk 78
global functions
calling from LabTalk 82
GraphLayer class 20
GraphObject class 21
GraphPage class 22
including LabTalk script 73
Layer class 23
Layout class 24
LayoutPage class 24
listing functions you can call from
LabTalk 82
matrix class 25
Matrix class 26
MatrixLayer class 27
MatrixObject class 27
MatrixPage class 28
Note class 29
operators
exponentiate 2
OriginObject class 29
Page class 29
PageBase class 30
pow function 2
progressBox class 30
Project class 31
ROIObject class 32
stdioFile class 32, 52
string class 33
StyleHolder class 34
vector class 36
waitCursor class 36
Worksheet class 37
WorksheetPage class 38
Origin projects
collections of windows 31
OriginObject class 29
Overloading functions 2
Page class 29
PageBase class 30
Passing arguments
calling functions from LabTalk 79
Path of current project (.OPJ)
LabTalk system variable 74
Path of Origin installation
LabTalk system variable 74
Percentiles 66
Plots See Graphs
Pointers 7
Polynomial regression 14, 61
Pragmas
xor
e exponentiate vs bitwise
exclusive OR 2
Preprocessor
pragmas
xor 2
Programming
ANSI C support 1
arrays 25, 36, 41
automation 67
basic data types 6
batch processing 67
C# support 5
C++ support 2
calling Fortran routines 48
calling NAG functions 62
categorical data 11, 12
collections of objects 12
columns in worksheet 13, 15
complex numbers 14, 43
creating and deleting objects 29
creating notes windows 29
data acquisition
GPIB 46
RS-232 43
data plots 14, 16
data sets 16
derived data types 7
differentiation 60
graph layers 23
graph page 22
graphic objects 21
graphs 16, 20
hourglass wait cursor 36
importing data 52, 69
including LabTalk in Origin C 73
integrating data 60
layers on page 29
layout pages 24
linear and polynomial regression 61
math operations on data sets 57
nonlinear curve fitting 58, 69
Origin matrices 26, 27
Origin projects 31
progress bar 30
Project Explorer 18
running application from Origin 83
statistics 65
style holders in graph 34
window operations 30
worksheet operations 37, 56
Progress bar
displaying 30
progressBox class 30
Project class 31
Project Explorer
accessing 18
accessing windows in a folder 18
creating folders 19
outputing subfolder names 18
Projects
collections of windows 31
name of current
LabTalk system variable 74
path of current
LabTalk system variable 74
Radians
LabTalk variable 73
Random access data file 18, 32, 52
References 7
Rescaling axes 20
ROIObject class 32
Rows (worksheets)
operations on 37
RS-232
data acquisition 43
Screen Reader
coordinates
LabTalk variable 73
Script files
programming 84
SelC1
worksheet selection range
LabTalk variable 73
SelC2
worksheet selection range
LabTalk variable 73
SelR1
worksheet selection range
LabTalk variable 73
SelR2
worksheet selection range
LabTalk variable 73
Sequential access data file 18, 32, 52
short type 6
signed type 6
Significant digits
for math operations
LabTalk variable 74
Statistics 65
linear and polynomial regression
61
stdioFile class 32, 52
Stream
importing and exporting data 18,
32, 52
string class 33
StringArray 33
Strings 40
returning string to LabTalk 80
string class 33
Structures 7
StyleHolder class 34
Syntax
ANSI C support 1
C# support 5
C++ support 2

T
Text label
deleting 23
Time
LabTalk variable 73
Toolbar buttons
programming 83, 84
Toolbars tab
Customize Toolbar dialog box 85

U
UINT type 6
unsigned type 6

V
Variables
declarations
C++ support 2
vector
returning vector to LabTalk 81
vector class 36
void type 6

W
waitCursor class 36
Windows
active
LabTalk system variable 74
collections in project 31
creating from template 30
naming/reNaming 30
returning the type 30
WORD type 6
Worksheet class 37
WorksheetPage class 38
Worksheets
column and row operations 37, 56
columns
counting 17
formatting 13
setting range 15
setting values 16, 56
creating from template 17, 30, 38,
56
opening from file 37
selection range
LabTalk variable 73
Workspace files
navigating in 93

X
X
tool coordinate
LabTalk variable 73
X1
initial scale value
LabTalk variable 73
X2
final scale value
LabTalk variable 73

Y
Y
tool coordinate
LabTalk variable 73
Y1
initial scale value
LabTalk variable 73
Y2
final scale value
Z

Z
  tool coordinate
  LabTalk variable 73
Z1
  initial scale value
  LabTalk variable 73
Z2
  final scale value
  LabTalk variable 73