

# Using the Quinn-Curtis .Net Software (QCChart2D and QCRTGraph) with Managed C++ (MC++)

August 9, 2005

Starting with **Visual Studio 2003**, you can create a Managed C++ application that interfaces to the Quinn-Curtis .Net software. The shell of the application is created using the VS 2003 **New Project** wizard, selecting **Visual C++ Projects** as the Project Type and **Windows Form Applications (.NET)** as the Template.

The advantage of using Managed C++ over ordinary C++, with or without MFC, is the automatic garbage collection that Managed C++ uses. No more unreleased pointers. Also, many other useful C#-like features become available, for example properties that do not need separate Set and Get methods, and multi-dimensional array indexing that uses a rectangular indexing `[*, *]` format rather than the C/C++ jagged pointer indexing `[*][*]` format.

In general, you should have a good working background in C#, C++ and MC++ before attempting to convert any of our examples to MC++. Here are some of the basic source level conversions that must be made. Note that most code in MC++ programs seems to be placed in the \*.h files. The \*.cpp files are empty or not even present. This makes translation from C# to MC++ even easier because you do not have to constantly copy prototype files from the classes \*.cpp file to the classes \*.h file.

## ***Make the classes that use the Quinn-Curtis .Net Libraries Managed C++ Classes***

The classes that access the QCChart2D and QCRTGraph libraries must use the Managed C++ `__gc` keyword, as below:

```
__gc public class SimpleBars
{
public:
    ChartView *chartVu;

public:
    SimpleBars(ChartView * chartvu)
    {
        chartVu = chartvu;
        InitializeChart();
    };
    .
    .
    .
}
```

```
}
```

## ***Convert all object references to pointers***

All C# objects created on the heap using the **new** keyword must be converted to pointers.

```
// C# Original Source
LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
yAxis.SetColor(Color.White);
chartVu.AddChartObject(yAxis);

TimeAxisLabels xAxisLab = new TimeAxisLabels(xAxis );
    xAxisLab.SetAxisLabelsFormat(ChartObj.TIMEDATEFORMAT_Y2000);
xAxisLab.SetColor(Color.White);
chartVu.AddChartObject(xAxisLab);
NumericAxisLabels yAxisLab = new NumericAxisLabels(yAxis);
yAxisLab.SetColor(Color.White);
    yAxisLab.SetAxisLabelsFormat(ChartObj.CURRENCYFORMAT);
chartVu.AddChartObject(yAxisLab);

AxisTitle yaxistitle = new AxisTitle( yAxis, theFont, "Millions $");
yaxistitle.SetColor(Color.White);
chartVu.AddChartObject(yaxistitle);

Grid ygrid = new Grid(xAxis, yAxis,ChartObj.Y_AXIS, ChartObj.GRID_MAJOR);
ygrid.SetColor(Color.White);
ygrid.SetLineWidth(1);
chartVu.AddChartObject(ygrid);

// Converted to MC++

LinearAxis * yAxis = new LinearAxis(pTransform1, ChartObj::Y_AXIS);
yAxis->SetColor(Color::White);
chartVu->AddChartObject(yAxis);

TimeAxisLabels * xAxisLab = new TimeAxisLabels(xAxis );
xAxisLab->SetAxisLabelsFormat(ChartObj::TIMEDATEFORMAT_Y2000);
xAxisLab->SetColor(Color::White);
chartVu->AddChartObject(xAxisLab);

NumericAxisLabels * yAxisLab = new NumericAxisLabels(yAxis);
yAxisLab->SetColor(Color::White);
yAxisLab->SetAxisLabelsFormat(ChartObj::CURRENCYFORMAT);
chartVu->AddChartObject(yAxisLab);

AxisTitle * yaxistitle = new AxisTitle( yAxis, theFont, "Millions $");
yaxistitle->SetColor(Color::White);
chartVu->AddChartObject(yaxistitle);

Grid * ygrid = new Grid(xAxis, yAxis,ChartObj::Y_AXIS, ChartObj::GRID_MAJOR);
```

```
ygrid->SetColor(Color::White);
ygrid->SetLineWidth(1);
chartVu->AddChartObject(ygrid);
```

Note that all method and property references have the “.” converted to the pointer reference “->”. Also note that all constant references have been converted from “.” to “::”. All of the conversions are straightforward.

## **Convert regular C/C++ arrays to Managed C++ arrays**

What is a little more complicated is the conversion of C# managed arrays to the equivalent MC++ code. The MC++ `__gc` keyword is again used and this time denotes that a specific array is to be treated as a managed, garbage collected array type based on the .Net Array class. While we have many examples of managed MC++ arrays of numeric, String, and other object types in the Bargraphs example program that is referenced in this article, our information originally came from a excellent Managed C++ tutorial, lesson 18, *Fundamentals of Arrays*.

<http://www.functionx.com/managedcpp/Lesson18.htm>

The syntax is very flexible and in our examples we use what we think is the most intuitive, though redundant, style.

```
// Original C# code

ChartCalendar []x1= new ChartCalendar[numpnts];
double [] y1= new double[numpnts];

// Converted to MC++

ChartCalendar * x1 __gc[] = new ChartCalendar * __gc[numpnts];
Double y1 __gc[] = new Double __gc [numpnts];
```

Note that **double** has been changed to the .Net value type **Double**, that the array brackets “[ ]”, prefixed with `__gc`, are placed after the variable name. Also the `ChartCalendar` object is converted to a pointer.

Arrays can be assigned to element by element using standard array indexing as below:

```
// From the SimpleBars.h example file

int i;

y1[0] = 100;
x1[0] = dynamic_cast<ChartCalendar *> ( currentdate->Clone());
```

```

currentdate->Add(ChartObj::MONTH,12);
for (i=1; i < nnumpts; i++)
{
    x1[i] = dynamic_cast<ChartCalendar *> (currentdate->Clone());
    y1[i] += y1[i-1] + (25 * i) * (0.75 -
        ChartSupport::GetRandomDouble());
    currentdate->Add(ChartObj::MONTH,12);
}

```

Note the **dynamic\_cast** instead of a regular C++ or C# cast. The compiler complains otherwise.

Two-dimensional arrays are a little more complicated. The trick is that managed MC++ arrays are more like C# arrays than C++ arrays.

```

// From the GroupBargraphs.h example file

int nNumPnts = 5, nNumGroups = 4;

ChartCalendar * xValues __gc[] = new ChartCalendar* __gc[nNumPnts];

Double groupBarData __gc[,] = new Double __gc[nNumGroups,nNumPnts];

theFont = new Font("Microsoft Sans Serif", 10, FontStyle::Bold);
xValues[0] = new ChartCalendar(1998, ChartObj::JANUARY, 1);
groupBarData[0,0] = 6.3; groupBarData[1,0] = 3.1;
groupBarData[2,0] = 2.2; groupBarData[3,0] = 1.8;

xValues[1] = new ChartCalendar(1999, ChartObj::JANUARY, 1);
groupBarData[0,1] = 5.8; groupBarData[1,1] = 4.3;
groupBarData[2,1] = 2.8; groupBarData[3,1] = 1.5;

xValues[2] = new ChartCalendar(2000, ChartObj::JANUARY, 1);
groupBarData[0,2] = 5.5; groupBarData[1,2] = 4.5;
groupBarData[2,2] = 2.5; groupBarData[3,2] = 2.1;

xValues[3] = new ChartCalendar(2001, ChartObj::JANUARY, 1);
groupBarData[0,3] = 4.1; groupBarData[1,3] = 5.4;
groupBarData[2,3] = 4.1; groupBarData[3,3] = 3.2;

xValues[4] = new ChartCalendar(2002, ChartObj::JANUARY, 1);
groupBarData[0,4] = 3.8; groupBarData[1,4] = 5.6;
groupBarData[2,4] = 4.3; groupBarData[3,4] = 3.3;

TimeGroupDataset * Dataset1 = new
    TimeGroupDataset("GroupTimeData", xValues, groupBarData);

```

Note that the 2D arrays are accessed using C#-like `*,*` indexing instead of the `[*][*]` C/C++ indexing style.

Arrays can be initialized one index at a time, as in the examples above, or all at once.

```
Double Male1960 __gc[] = {19.5, 15.1, 8.2, 3.1, 0.2};
Double Female1960 __gc[] = {18.5, 15.5, 9.2, 4.1, 0.4};
Double Male2002 __gc[] = {13.1, 17.7, 17.3, 11.5, 1.5};
Double Female2002 __gc[] = {12.1, 18.1, 18.2, 12.1, 3.4};

Double SpaceSpending __gc[] =
    {0.5,1.0,1.5,2.5,4.0,5.0,5.9,5.6,4.8,4.2,3.8,
     3.4,3.4,3.4,3.4,3.4,3.8,4.0,4.1,4.3,4.9,5.5,6.0,6.5,7.3,7.5,7.8,
     8.0,9.0,10.1,12.5,14,14.1,14.3, 14.1,13.8,14.2,14.3,14.2,14.1,
     13.8,14.2,14.5,14.8,15.3,16.0};

String *CountryNames __gc[] =
    {"", "China", "Japan", "Russia", "Italy", "Sweden",
     "Denmark", "Mexico", "Brazil", "France",
     "Australia", "Spain", "United States", "England"};
```

### ***Move any variable initializations from class variable declaration area to an initialization method***

MC++ only allows static variables to be initialized in the class variable declaration section of the \*.h header file. Continue to declare the variables in the usual way, but move the initialization to a separate method.

```
// C# code
public class AnExample
{
    int count = 0;
    int numChannels = 8;

    RTPProcessVar [] AnnunciatorProcessItems = new RTPProcessVar[16];

    AnExample ()
    {
        InitializeChart();
    };
    .
    .
    .
}
```

```

// MC++ code
__gc public class AnExample
{
    int count;
    int numChannels;

    RTProcessVar * AnnunciatorProcessItems __gc[];

Public:
    AnExample ()
    {
        InitializeInstanceVariables();
        InitializeChart();
    };
    .
    .
    .
private:
    void InitializeInstanceVariables()
    {
        count = 0;
        numChannels = 8;
        AnnunciatorProcessItems = new RTProcessVar __gc[16];
    }
    .
    .
    .
}

```

## ***Convert Event Handlers***

The Visual Studio .Net documentation on delegates and event handlers does not give any examples for MC++. The prototypes of the delegates must have a specific signature to avoid compile errors. While the QCChart2D Charting Tools for .Net does not use event handlers, the QCRTGraph Real-Time Graphics Tools for .Net uses them extensively, for processing buttons, trackbars, timers and alarm events. See the example program **ProcessMonitoring** in the Quinn-Curtis\DotNet\QCRTGraph\Visual MCPP\Examples\ProcessMonitoring folder.

Here are a few examples:

### **RTControlButton**

```

// Set the event delegate for StartControl
StartControl->Click +=
    new System::EventHandler(this, controlOn_Button_Click);
.

```

```

.
.

//Event delegate for StartControl
private: System::Void controlOn_Button_Click(Object * sender,
        System::EventArgs * e)
{
    .
    .
    .
}

```

## RTControlTrackbar

```

// Set event delegate for the proportionalControlTrackBar trackbar
proportionalControlTrackBar->Click +=
    new System::EventHandler(this, pidTrackBarParameter_Click);
.
.
.

//Event delegate for pidTrackBarParameter
private: System::Void pidTrackBarParameter_Click(Object * sender,
        System::EventArgs * e)
{
    .
    .
    .
}

```

## System::Timers::Timer

```

timer1 = new System::Timers::Timer();
// Set event delegate for the timer1 timer
timer1->Elapsed +=
    new System::Timers::ElapsedEventHandler(this, timer1_Tick);
timer1->Interval = 500;
.
.
.

//Event delegate for timer1
private: System::Void timer1_Tick(System::Object * sender,
        System::Timers::ElapsedEventArgs * e)

```

```
{  
    .  
    .  
    .  
}
```

Note that the delegates are private, and that the return type is **System::Void**

### ***Add the QCChart2D ChartView to the Visual Studio Toolbox***

This covers the most important aspects of programming the Quinn-Curtis .Net software using MC++. There is still a secondary issue associated with getting the **ChartView** class that the graph is drawn in, into a MC++ form. This is complicated by the strange inconsistency between (C# and VB) versus MC++. In our standard documentation, we describe how to create a new class for your project that derives from the .Net **UserControl** class. Since our **ChartView** class subclasses and extends the standard .Net **UserControl** class, we just have you change the parent class of your new **UserControl** class, to **ChartView**. Once initially compiled, the new **ChartView** derived class appears in the Toolbox and can be added to any of the forms, tabs or panels of the current project. Unfortunately, MC++ does NOT allow you to add a **UserControl** class to your project, even though that option is present when you invoke the **Add Class** wizard. Instead it gives you some cryptic error message about how UserControls can only be added to DLL projects. Apparently, there is some issue with verification of the **UserControl** class that the MC++ cannot handle.

So, we use the following technique for adding a custom chart to a form, panel or tab. Since our **ChartView** class, located in our QCChart2DNet DLL, is a **UserControl** derived class, it CAN be added to the Toolbox directly. From the VS 2003 main menu, select **Tools | Add/Remove Toolbox Items**, and browse to and add **QCChart2DNet.DLL**, located in **the Quinn-Curtis\DotNet\lib** subdirectory. This should place the **ChartView** class in the Toolbox under the **MyUserControls** heading.

Adding and instance of the **ChartView** object from the Toolbox will automatically add the QCChart2DNet.DLL to the project references. If you use the QCRTGraph for .Net routines (they are easy to spot since that all start with RT), you must explicitly add the QCRTGraphNet.DLL to the project references using the projects **Add Reference** Wizard, and browsing to the QCRTGraphNet.DLL in the Quinn-Curtis\DotNet\lib folder.

## **Add the ChartView to the Form and initialize the Chart using a new class**

Now you should be able to add an instance of the **ChartView** class to any form, panel or tab that you have. What needs to be done next is to create a new class that adds the necessary QCChart2D, or QCRTGraph objects to the **ChartView**. This chart initializing class that you create does not have to be derived from anything. It does have to use the **\_\_gc** keyword, and include the necessary using statements.

```
#pragma once

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Diagnostics;
using namespace System::Drawing;
using namespace System::Drawing::Drawing2D;
using namespace com::quinncurtis::chart2dnet;

namespace Bargraphs
{
    /// <summary>
    /// Summary for SimpleBars
    /// </summary>
    __gc public class SimpleBars
    {
    public:
        ChartView *chartVu;

    public:
        SimpleBars(ChartView * chartvu)
        {
            chartVu = chartvu;
            InitializeChart();
        };
        ~SimpleBars(void)
        {
        };

        void InitializeChart() {

        .
        .
        .

        };
    };
}
```

Unfortunately, the VS 2003 **Add Class** wizards do not have one for a generic MC++ class that you can use. Instead you must create a generic C++ class and modify it to

include the **namespace**, **\_\_gc** keyword, **includes** and **using** statements that your project needs.

See the SimpleBars.h of the Bargraphs project in the download below for a complete example.

The constructor to this class passes in the already instantiated **ChartView** object, saves it, and then proceeds to add graph objects to it to make the chart.

## **Examples**

Download the file

<http://www.quinn-curtis.com/QCNetMCPPEXamples.zip>

for the examples. The examples are:

**ScatterPoints** – A QCChart2D example that creates a simple line and scatter plot. It also demonstrates printing and saving a chart as an image file. Located at Quinn-Curtis\DotNet\QCChart2D\Visual MCPP\Examples\ScatterPoints

**Bargraphs** – A QCChart2D example that displays 11 different bar charts created using a variety of techniques, organized using a tab control. Located at Quinn-Curtis\DotNet\QCChart2D\Visual MCPP\Examples\BarGraphs.

**ProcessMonitoring** – A QCRTGraph example that combines dynamic bar indicators, meters, annunciators, scrolling line plots, buttons and trackbars in a single complex graph that simulates a pilot plant control system. Located at Quinn-Curtis\DotNet\QCRTGraph\Visual MCPP\Examples\ProcessMonitoring.

## **Important Note**

You must have the **QCChart2D for .Net** software, either the **Trial** or **Developers Version**, installed and working before you can compile and run the examples. Make sure you compile the examples before you start trying to view the files. We delete the 10M of temporary files that Visual Studio keeps in the project before we compress the download and Visual Studio needs to recreate these before you can view the projects forms. Otherwise, Visual Studio has the nasty habit of just deleting everything on the main form, without telling you.

## ***Feedback***

This is draft #2 of this application note. We are interested in your feedback. Please post any questions or corrections that you might have on **Tools for Microsoft .Net & .Net Compact Framework** forum at:

<http://www.quinn-curtis.com/ForumFrame.htm>