

Visual Studio Native Type Visualization in VS 11

Introduction

In VS 11, Visual Studio native debugger introduces a new type visualization framework that allows customizing the way Visual Studio displays native types in debugger variable windows (watch, locals, data tips etc.). It supersedes the autoexp.dat file that has been used in earlier versions of Visual Studio and offers xml syntax, better diagnostics, versioning and multiple file support. Component authors can use this framework to create visualization rules for their types making it easy for developers to inspect them during debugging.

As an example, the image below shows how a variable of type `Windows::UI::Xaml::Controls::TextBox` is displayed under the debugger without any custom visualizations applied. The highlighted row points to the data member supplying the “Text” property of the `TextBox` class. The complex class hierarchy makes it difficult to find the value of this important member as we have to drill down deep in the variable window. Moreover, the debugger doesn’t know how to interpret the custom string type used by the object so we cannot see the string held inside the textbox.

myTextBox	0x05679e28 {...}	Windows::UI::Xaml::Controls::TextBox
[ctl::ComObject<DirectUI::TextBox>]	{...}	ctl::ComObject<DirectUI::TextBox>
ctl::InternalComObject<DirectUI::TextBox>	{...}	ctl::InternalComObject<DirectUI::TextBox>
DirectUI::TextBox	{m_pDeleteButton=0x00000000 {m_pointerPosition: DirectUI::TextBox	DirectUI::TextBox
DirectUI::TextBoxGenerated	{...}	DirectUI::TextBoxGenerated
+ Windows::UI::Xaml::Controls::IControl	{...}	Windows::UI::Xaml::Controls::IControl
+ DirectUI::Control	{m_suspendStateChanges=0 '\0' }	DirectUI::Control
DirectUI::ControlGenerated	{...}	DirectUI::ControlGenerated
+ Windows::UI::Xaml::Controls::IControl	{...}	Windows::UI::Xaml::Controls::IControl
+ Windows::UI::Xaml::Controls::IControlOverrides	{...}	Windows::UI::Xaml::Controls::IControlOverrides
+ Windows::UI::Xaml::Controls::IControlProtected	{...}	Windows::UI::Xaml::Controls::IControlProtected
+ DirectUI::FrameworkElement	{m_pDataContextChangedSource=0x00000000 {...} }	DirectUI::FrameworkElement
DirectUI::FrameworkElementGenerated	{...}	DirectUI::FrameworkElementGenerated
+ Windows::UI::Xaml::IFrameworkElement	{...}	Windows::UI::Xaml::IFrameworkElement
+ Windows::UI::Xaml::IFrameworkElementOverrid	{...}	Windows::UI::Xaml::IFrameworkElementOverrid
+ DirectUI::UIElement	{m_pAP=0x00000000 {...} m_IsLocationValid=0 '\0' r	DirectUI::UIElement
DirectUI::UIElementGenerated	{...}	DirectUI::UIElementGenerated
+ Windows::UI::Xaml::UIElement	{...}	Windows::UI::Xaml::UIElement
+ Windows::UI::Xaml::UIElementOverrid	{...}	Windows::UI::Xaml::UIElementOverrid
+ DirectUI::DependencyObject	{m_EventMap={size = 0} m_pPropertyValueReferen	DirectUI::DependencyObject
+ Windows::UI::Xaml::IDependencyObj	{...}	Windows::UI::Xaml::IDependencyObj
+ ctl::WeakReferenceSource	{m_pWeakReference=0x00000000 {m_pRef=??? } }	ctl::WeakReferenceSource
+ DirectUI::IReferenceTracker	{...}	DirectUI::IReferenceTracker
+ DirectUI::IReferenceTrackerInternal	{...}	DirectUI::IReferenceTrackerInternal
+ m_EventMap	{size = 0}	std::map<int,DirectUI::IUn
+ m_pPropertyValueReferences	0x0567a060 {size = 2}	std::map<DirectUI::Depende
+ m_pltemReferences	0x00000000 {...}	std::list<DirectUI::Dependen
+ m_pDO	0x056484c8 {m_plnheritedProperties=0x00000000 {n	CDependencyObject *
[CTextBox]	{m_plnInputScope=0x00000000 {m_pNames=??? } m_	CTextBox
+ CTextBoxBase	{m_iMaxLength=0 m_bIsReadOnly=0 m_bAcceptsR	CTextBoxBase
+ m_plnInputScope	0x00000000 {m_pNames=??? }	CInputScope *
+ m_pText	0x0565b920 {...}	CXString *
+ XSTRING_BASE	{cString=15 pString=0x0565b931 {30976} flags=0 }	XSTRING_BASE

The same object with custom visualization rules applied for the type has a much simpler view in the variable window as shown below. The important members of the class can be viewed together and the debugger is able to show the underlying string value of the custom string type.

myTextBox	0x06c4c148 {Name = L"myTextBox" Text = L"My TextBox Text"}	Windows::UI::Xaml::Controls::
Text	0x06c2dd68 L"My TextBox Text"	CXString *
SelectionLength	0	int
SelectionStart	0	int
SelectedText	0x00000000 ???	CXString *
IsSpellCheckEnabled	false	bool
IsTextPredictionEnabled	true	bool
InputScope	0x00000000 {m_pNames=??? }	CInputScope *
MaxLength	0	int
IsReadOnly	false	bool
AcceptsReturn	false	bool
TextWrapping	NoWrap (1)	TextWrapping
TextAlignment	TextAlignmentLeft (1)	TextAlignment
IsTabStop	true	bool

Creating visualizations for new types

Natvis files

Visualizers for native types are specified in .natvis files. A natvis file is simply an xml file (with a .natvis extension) with its schema defined in <VSINSTALLDIR>\Xml\Schemas\natvis.xsd. Visual Studio ships with a few natvis files in <VSINSTALLDIR>\Common7\Packages\Debugger\Visualizers folder. These files contain visualization rules for many common types and can serve as examples when writing visualizers for new types.

The basic structure of a natvis file is as follows, where each **Type** element represents a visualizer entry for a type whose fully qualified name is specified in the **Name** attribute.

```
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/2010">
  <Type Name="MyNamespace::CFoo">
    .
    .
  </Type>

  <Type Name="...">
    .
    .
  </Type>
</AutoVisualizer>
```

You can start writing a visualizer for your types by creating a natvis file having the above structure and dropping it into one of the locations below:

- %VSINSTALLDIR%\Common7\Packages\Debugger\Visualizers (requires admin access)
- %USERPROFILE%\My Documents\Visual Studio Dev11\Visualizers\
- VS extension folders (Extension manifest must contain a "NativeVisualizer" asset entry)

At the start of each debugging session, Visual Studio will load and process every natvis file it can find in these locations (it is NOT necessary to restart Visual Studio). This makes writing new visualizers easy as you can stop debugging, make changes to your visualizer entries, save the natvis file and start debugging again to see the effects of your changes.

Natvis diagnostics

Natvis diagnostics is a very important tool that helps troubleshooting issues when writing new visualizers. When VS debugger encounters errors in a visualizer entry (e.g. xml schema errors, expression fails to parse), it will simply

ignore it and display the type in its raw form (or pick another suitable visualizer). To understand why a certain visualizer entry is ignored and to see what the underlying errors are, you can turn on visualization diagnostics which is controlled by the following registry value:

```
[HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\11.0_Config\Debugger]
```

```
"EnableNatvisDiagnostics"=dword:00000001
```

You can import the following registry file which automatically does this for you:



EnableNatvisDiagnostics.reg

When turned on, you will see diagnostic status messages (e.g. when a natvis file is parsed, an expression is successfully evaluated) and error messages (e.g. file parse errors, expression parse errors) in the **output window** in Visual Studio.

Syntax Reference

While the full syntax of a visualizer entry can be found in the schema file mentioned above, the structure of a basic visualizer entry looks like:

```
<Type Name="[fully qualified type name]">
  <DisplayString Condition="[Boolean expression]">[Display value]</DisplayString>
  <Expand>
    ...
  </Expand>
</Type>
```

where we specify:

- What type this visualizer should be used for
- What the value of an object of that type should look like
- What the children of the object should look like when user expands it in the variable window

Condition attribute

The optional condition attribute is available for many visualizer elements and specifies when a visualization rule should be used. If the expression inside the condition attribute is false, then the visualization rule specified by the element is not applied. If it's evaluated to true, or if there is no condition attribute then the visualization rule is applied to the type. You can use this attribute to have if-else logic in the visualization entries. For instance, the visualizer below defines two DisplayString elements (explained further below) for a smart pointer type:

```
<Type Name="std::auto_ptr<*>">
  <DisplayString Condition="_Myptr == 0">empty</DisplayString>
  <DisplayString>auto_ptr {*_Myptr}</DisplayString>
  <Expand>
    <ExpandedItem>_Myptr</ExpandedItem>
  </Expand>
</Type>
```



When `_Myptr` member is null, the condition of the first DisplayString element will be true, therefore that element takes effect. When the `_Myptr` member is not null, the condition evaluates to false and the second DisplayString element takes effect.

DisplayString

DisplayString node specifies the string to be shown as the value of the variable. It accepts arbitrary strings mixed with expressions. Everything inside curly braces is interpreted as an expression and gets evaluated. For instance:

```
<Type Name="CPoint">
  <DisplayString>{{x={x} y={y}}}</DisplayString>
</Type>
```

will result in variables of type CPoint to look like:

Name	Value	Type
  point	{x=3 y=5}	CPoint

Here, x and y, which are members of CPoint, are inside curly braces and their values are evaluated. The example also shows that you can escape a curly brace by using double curly braces (i.e. {{ or }}).



One important point to remember is DisplayString element is the only element that accepts arbitrary strings and the curly brace syntax. All other visualization elements accept only expressions that are evaluated by the debugger.

StringView

StringView element defines the expression whose value is going to be sent to the built-in text visualizers. For instance, suppose we have the following visualizer for the ATL::CStringT type:

```
<Type Name="ATL::CStringT<wchar_t,*>">
  <DisplayString>{m_pszData,su}</DisplayString>
</Type>
```




which will cause CStringT objects to look like below:

Name	Value	Type
  string	L"This is a string"	ATL::CStringT<wchar_t,*>

Adding a StringView element will indicate to the debugger that this value can be viewed by a text visualizer:

```
<Type Name="ATL::CStringT<wchar_t,*>">
  <DisplayString>{m_pszData,su}</DisplayString>
  <StringView>m_pszData,su</StringView>
</Type>
```

Notice the glass icon shown next to the value below. Clicking the icon will launch the text visualizer which will display the string that m_pszData points to.

Name	Value	Type
  string	L"This is a string"	 ATL::CStringT<wchar_t,*>

Expand

Expand node is used to customize the children of the visualized type when the user expands it in the variable windows. It accepts a list of child nodes, which in turn define the child elements. Before we look at the xml nodes

allowed under it, it is important to know that Expand node is optional and if no Expand is specified in a visualizer entry, Visual Studio's default expansion rules will be used. If an expand node is specified with no child nodes under it, then the type won't be expandable in the debugger windows (i.e. no plus sign next to the variable name).

Item Expansion

The **Item** node, which is the most basic and most common node to be used under an Expand node, defines a single child element. For instance, if you have a CRect class with top, left, right, bottom as its fields and the following visualizer entry

```
<Type Name="CRect">
  <DisplayString>{{top={top} bottom={bottom} left={left} right={right}}}</DisplayString>
  <Expand>
    <Item Name="Width">right - left</Item>
    <Item Name="Height">bottom - top</Item>
  </Expand>
</Type>
```

then CRect type is going to look like

Name	Value	Type
[-] rect	{top=200 bottom=300 left=100 right=600}	CRect
[-] Width	500	long
[-] Height	100	long
[-] [Raw View]	0x00d3f6f4 {...}	CRect *
[-] tagRECT	{left=100 top=200 right=600 ...}	tagRECT
[-] left	100	long
[-] top	200	long
[-] right	600	long
[-] bottom	300	long

The expressions specified in Width and Height elements are evaluated and shown in the value column. [Raw View] node is automatically created by the debugger for the user anytime a custom expansion is used. It is expanded in the screenshot above to show how the raw view of the object is different than its visualization. Note that the default expansion used by VS creates a subtree for the base class and lists all the data members of a class as the children.

An additional point to remember is if the expression of the item element points to a complex type the Item node itself will be expandable.

ArrayItems Expansion

ArrayItems node can be used to have VS debugger interpret the type as an array and display its individual elements. The visualizer for std::vector is a good example using this node:

```
<Type Name="std::vector<*>">
  <DisplayString>{{size = {_Mylast - _Myfirst}}}</DisplayString>
  <Expand>
    <Item Name="[size]">_Mylast - _Myfirst</Item>
    <Item Name="[capacity]">(_Myend - _Myfirst)</Item>
    <ArrayItems>
      <Size>_Mylast - _Myfirst</Size>
      <ValuePointer>_Myfirst</ValuePointer>
    </ArrayItems>
  </Expand>
</Type>
```

A `std::vector` shows its individual elements when expanded in the variable window:

Name	Value	Type
myvector	{size = 3}	std::vector<int,std::allocator<int> >
[size]	3	int
[capacity]	3	int
[0]	1	int
[1]	2	int
[2]	3	int

At a minimum, the `ArrayItems` node must have the **Size** expression (which must evaluate to an integer) for the debugger to understand the length of the array and the **ValuePointer** expression that should point to the first element (which must be a pointer of the element type that is not `void*`). The array lower bound is assumed to be 0 which can be overridden by using **LowerBound** node (examples of this can be found in the default `natvis` files shipped with VS).

Multi-dimensional arrays can also be specified. The debugger needs just a little bit more information to properly display child elements in that case:

```
<Type Name="Concurrency::array<*,*>">
  <DisplayString>extent = {_M_extent}</DisplayString>
  <Expand>
    <Item Name="extent">_M_extent</Item>
    <ArrayItems Condition="_M_buffer_descriptor._M_data_ptr != 0">
      <Direction>Forward</Direction>
      <Rank>$T2</Rank>
      <Size>_M_extent._M_base[$i]</Size>
      <ValuePointer>($T1*) _M_buffer_descriptor._M_data_ptr</ValuePointer>
    </ArrayItems>
  </Expand>
</Type>
```

Direction specifies whether the array is row-major or column-major order. **Rank** specifies the rank of the array. **Size** element accepts the implicit `$i` parameter which it substitutes with dimension index to find the length of the array in that dimension. For instance, in the example above the expression `_M_extent._M_base[0]` should give the length of the 0th dimension, `_M_extent._M_base[1]` the 1st and so on.

Here's how a two dimensional `Concurrency::array` object looks like in the debugger:

myAmpCircle	extent = (16, 16)	Concurrency::array<int,2> &
[Dimensions]	[16, 16]	
[0,0]	1	int
[0,1]	2	int
[0,2]	3	int
[0,3]	4	int
[0,4]	5	int
[0,5]	6	int
[0,6]	0	int
[0,7]	0	int
[0,8]	0	int
[0,9]	0	int

IndexListItems Expansion

ArrayItems assume array elements are laid out contiguously in memory. Debugger gets to the next element by simply incrementing its pointer to the current element. To support cases where you need to manipulate the index to the value node, index list items can be used. Here's a visualizer using **IndexListItems** node:

```
<Type Name="Concurrency::multi_link_registry<*>">
  <DisplayString>{{size = {_M_vector._M_index}}}</DisplayString>
  <Expand>
    <Item Name="[size]">_M_vector._M_index</Item>
    <IndexListItems>
      <Size>_M_vector._M_index</Size>
      <ValueNode>*(_M_vector._M_array[$i])</ValueNode>
    </IndexListItems>
  </Expand>
</Type>
```

The only difference between ArrayItems and IndexListItems is that the ValueNode expects the full expression to the i^{th} element with the implicit **\$i** parameter.

LinkedListItems Expansion

If the visualized type represents a linked list, debugger can be instructed to display its children via **LinkedListItems** node. Here's the visualizer for the CATList type using this feature:

```
<Type Name="ATL::CatList<*,*>">
  <DisplayString>{{Count = {m_nElements}}}</DisplayString>
  <Expand>
    <Item Name="Count">m_nElements</Item>
    <LinkedListItems>
      <Size>m_nElements</Size>
      <HeadPointer>m_pHead</HeadPointer>
      <NextPointer>m_pNext</NextPointer>
      <ValueNode>m_element</ValueNode>
    </LinkedListItems>
  </Expand>
</Type>
```

Size expression refers to the length of the list, **HeadPointer** points to the first element, **NextPointer** refers to the next element, and **ValueNode** refers to the value of the item. Two important points to remember with LinkedListItems node are: 1) NextPointer and ValueNode expressions are evaluated under the context of the linked list node element and not the parent list type. In the example above, CATList has a CNode class (can be found in atcoll.h) that represents a node of the linked list. m_pNext and m_element are fields of that CNode class and not of CATList class. 2) ValueNode can be left empty or have "this" to refer to the linked list node itself.

TreeItems Expansion

If the visualized type represents a tree, debugger can walk the tree and display its children via **TreeItems** node. Here's the visualizer for the std::map type using this feature:

```
<Type Name="std::map<*>">
  <DisplayString>{{size = {_Mysize}}}</DisplayString>
  <Expand>
    <Item Name="[size]">_Mysize</Item>
    <Item Name="[comp]">comp</Item>
    <TreeItems>
      <Size>_Mysize</Size>
      <HeadPointer>_Myhead-&gt;_Parent</HeadPointer>
```

```

        <LeftPointer>_Left</LeftPointer>
        <RightPointer>_Right</RightPointer>
        <ValueNode Condition="!((bool)_Isnll)">_Myval</ValueNode>
    </TreeItems>
</Expand>
</Type>

```

The syntax is very similar to LinkedListItems node. **LeftPointer**, **RightPointer**, **ValueNode** are evaluated under the context of the tree node class, and **ValueNode** can be left empty or have “this” to refer to the tree node itself.

ExpandedItem Expansion

ExpandedItem can be used to generate an aggregated children view by having properties coming from base classes or data members displayed as if they were children of the visualized type. The specified expression is evaluated and the child nodes of the result are appended to the children list of the visualized type. For instance, suppose we have a smart pointer type `auto_ptr<vector<int>>` which will normally be displayed as:

Name	Value	Type
mptr,! <ul style="list-style-type: none"> _Myptr <ul style="list-style-type: none"> [size] [capacity] [0] [1] [2] 	{ _Myptr=0x00d9fcdc {size = 3} } 0x00d9fcdc {size = 3} 3 3 1 2 3	std::auto_ptr<std::vector<int,std::allocator<int> > > std::vector<int,std::allocator<int> > * int int int int int

To see the values of the vector, we need to drill down two levels in the variable window passing through `_Myptr` member. Adding a visualizer entry using **ExpandedItem** element:

```

<Type Name="std::auto_ptr<*>">
    <DisplayString>auto_ptr {*_Myptr}</DisplayString>
    <Expand>
        <ExpandedItem>_Myptr</ExpandedItem>
    </Expand>
</Type>

```

we can eliminate `_Myptr` variable from the hierarchy and directly view vector elements:

mptr <ul style="list-style-type: none"> [size] [capacity] [0] [1] [2] [Raw View] 	auto_ptr {size = 3} 3 3 1 2 3 0x0031fb30 { _Myptr=0x0031fb64 {s	std::auto_ptr<std::vector<int,std::allocator<int> > > int int int int int std::auto_ptr<std::vector<int,std::allocator<int> > > *
--	---	---

The example below shows how to aggregate properties from the base class in a derived class. Suppose `CPanel` class derives from `CFrameworkElement`. Instead of repeating the properties that come from the base `CFrameworkElement` class, the **ExpandedItem** node allows those properties to be appended to the children list of `CPanel` class. The **nd** format specifier which turns off visualizer matching for the derived class is necessary here. Otherwise, the expression “*(CFrameworkElement*)this” will cause the “CPanel” visualizer to be applied again as under the default visualizer type matching rules it is most appropriate one. With **nd** switch base class visualizer is used or if the base class doesn’t have any visualizer default expansion is used.


```

<Type Name="CPanel">
  <DisplayString>{{Name = {(m_pstrName)}}}</DisplayString>
  <Expand>
    <Item Name="IsItemsHost">(bool)m_bItemsHost</Item>
    <ExpandedItem>*(CFrameworkElement*)this,nd</ExpandedItem>
  </Expand>
</Type>

```

Synthetic Item Expansion

Where ExpandedItem element provides a flatter view by eliminating hierarchies, **Synthetic** node does the opposite. It allows one to create an artificial child element (i.e. one that is not a result of an expression) which might contain children elements on its own. In the example below, the visualizer for the Concurrency::array type uses synthetic node to show a diagnostic message to the user:

```

<Type Name="Concurrency::array<T>*>">
  <DisplayString>extent = {_M_extent}</DisplayString>
  <Expand>
    <Item Name="extent" Condition="_M_buffer_descriptor._M_data_ptr == 0">_M_extent</Item>
    <ArrayItems Condition="_M_buffer_descriptor._M_data_ptr != 0">
      <Rank>$T2</Rank>
      <Size>_M_extent._M_base[$i]</Size>
      <ValuePointer>($T1*) _M_buffer_descriptor._M_data_ptr</ValuePointer>
    </ArrayItems>
    <Synthetic Name="Array" Condition="_M_buffer_descriptor._M_data_ptr == 0">
      <DisplayString>Array members can be viewed only under the GPU debugger</DisplayString>
    </Synthetic>
  </Expand>
</Type>

```

myAmpCircle	extent = (16, 16)	Concurrency::array<int,2>
extent	(16, 16)	Concurrency::extent<2>
Array	Array members can be viewed only under the GPU debugger	

Visualizer – Type Matching

Here are the general rules governing how visualizers are matched with types to be viewed in the debugger windows:

- A visualizer entry is applicable for the type specified in its name attribute AND for all the types that are derived from it. If there is a visualizer for both a base class and the derived class, the derived class visualizer takes precedence.
- Name attribute of the Type element accepts “*” as a wildcard character that can be used for templated class names:

```

<Type Name="ATL::CATLArray<T>*>">
  <DisplayString>{{Count = {m_nSize}}}</DisplayString>
</Type>

```

Here, the same visualizer will be used whether the object is a CATLArray<int> or a CATLArray<float>. If there is another visualizer entry for a “CATLArray<float>” then it takes precedence over the generic one.

Note that template parameters can be referenced in the visualizer entry by using macros **\$T1**, **\$T2**... Please see the visualizer files shipped with VS to find examples of these.

- If a visualizer entry fails to parse and validate than the next available visualizer is used.

Versioning

Visualizers can be scoped to specific modules and their versions so that name collisions can be minimized and different visualizers can be used for different versions of the types. The optional version element is used to specify this. For instance:

```
<Type Name="DirectUI::Border">
  <Version Name="Windows.UI.Xaml.dll" Min="1.0" Max="1.5"/>
  <DisplayString>{{Name = {*(m_pDO-&gt;m_pstrName)}}}</DisplayString>
  <Expand>
    <ExpandedItem>*(CBorder*)(m_pDO)</ExpandedItem>
  </Expand>
</Type>
```

Here, the visualizer is only applicable for the DirectUI::Border type found in the Windows.UI.Xaml.dll from version 1.0 to 1.5. Note that while adding version elements will scope visualizer entry to a particular module / version and reduce inadvertent matches, if a type is defined in a common header file that is used by different modules, it will prevent the visualizer from taking effect when the type is not in the specified module.

TODO:

HRESULT elements

UIVisualizer elements

Natvis validator