Complex Types

Besides integral types, such as various integer and floating-point types, also complex types are supported. This is achieved by converting the complex type into an array of basic types. These can be static or dynamic in size.

To make this possible, the code generator must know the underlying basic-types for a specific complex type which is done by providing an optional Complex-Type definition file. Further the user must implement matching conversion functions between the specified basic and complex types.

Note: Complex-Types using mixed data types can be supported by treating them as simple byte-arrays

- XML Definition
- Converter
- Editor

XML Definition

Complex Types are defined in an XML Schema file which is located at “cgi_studio_connector/schema/ComplexType.xsd”. This file can be used to validate xml files and with some editors (e.g. Visual-Studio) and provide code-completion when writing the XML file.

ComplexTypeDefinition-Element

Root Element. List of Complex types.

Attributes:
None

Children:

Argument	Occurrence	Description
ComplexType	0 - *	Complex type

ComplexType-Element

Defines a specific Complex type.

Attributes:

Argument	Occurrence	Description
native_type	Yes	Native data type (e.g. Candera::Vector3)
basic_type	Yes	Corresponding basic type. Valid values are: - Double - Single - Int8 - UInt8 - Int16 - UInt16 - Int32 - UInt32 - Int64 - UInt64 - Bool
basic_type_length	Yes	Length of basic type -1: Dynamic size >0: Fixed size

default_value	Yes	Default value of type

Children:
None

Sample

Below a sample XML file is shown

```
<?xml version="1.0"?>
<ComplexTypeDefinition xsi:noNamespaceSchemaLocation="../../tools/Core/Schema/ComplexType.xsd"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ComplexType native_type="Candera::Vector2" basic_type="Single" basic_type_length="2"
                        default_value="Candera::Vector2(0.0F, 0.0F)"/>
   <ComplexType native_type="Candera::Vector3" basic_type="Single" basic_type_length="3"
                        default_value="Candera::Vector3(0.0F, 0.0F, 0.0F)"/>
   <ComplexType native_type="FeatStd::String" basic_type="UInt8" basic_type_length="-1"
                        default_value="FeatStd::String(&amp;quot;&amp;quot;)"/>
   <ComplexType native_type="Custom::ByteArray" basic_type="UInt8" basic_type_length="-1"
                        default_value=""/>
</ComplexTypeDefinition>
```

Converter

For conversion between the basic type and actual complex type, conversion functions needs to be available. This function must be implemented inside the “ComplexType” namespace (additionally to the default namespace for the Interface) and within the “ComplexType.h/cpp” files.

For fixed size data types the conversion looks like the following:

```
void Convert(FeatStd::Float* from, Candera::Vector3* to)
{
   to->SetX(from[0]);
   to->SetY(from[1]);
   to->SetZ(from[2]);
}

void Convert(Candera::Vector3* from, FeatStd::Float* to)
{
   to[0] = from->GetX();
   to[1] = from->GetY();
   to[2] = from->GetZ();
}
```

For login/tracing purposes a “PrintValue_CUSTOM_TYPE” function is also required. This function’s purpose is to print the given value to the provided buffer. The provided value is the basic type not the final complex type (e.g. float[3] instead of Candera::Vector3). The function must return the bytes written to the buffer.

```
FeatStd::Int PrintValue_Candera_Vector3(void* buffer, FeatStd::UInt32 bufferLength, const void* value)
{
   const Candera::Float* floatValues = (const Candera::Float *)value;
   return FeatStd::Internal::String::StringPrintf((char*)buffer, bufferLength,
                  "[%.2f, %.2f, %.2f]", floatValues[0], floatValues[1], floatValues[2]);
}
```

For dynamic arrays (such as used for strings) the “DataBuffer” class will be used when the type is used in Data-Interfaces and the “Queue” class will be used if this type is used in actions or events. Additionally, a function that converts a void* with size information to the target type is required in both cases.

```
void Convert(FeatStd::String* from, Internal::DataBuffer* to)
{
   to->CopyFrom(from->GetCString(), from->GetCharCount());
}

void Convert(FeatStd::String* from, Internal::Queue* to)
{
   to->WriteSizeData(from->GetCString(), from->GetCharCount());
}

void Convert(const void* data, FeatStd::UInt32 dataSize, FeatStd::String* to)
{
   FEATSTD_UNUSED(dataSize);

   *to = FeatStd::String((FeatStd::TChar*)(data));
}

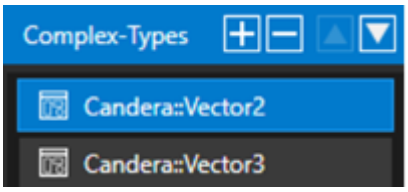
FeatStd::Int PrintValue_FeatStd_String(void* buffer, FeatStd::SizeType bufferLength,
                                       const void* value, FeatStd::UInt32 valueSize)
{
   return FeatStd::Internal::String::StringPrintf((char*)buffer, bufferLength, "%s", (const char*)value);
}
```

Editor

An editor for the Native-Function definition can be found here:
“cgi_studio_connector/bin/ComplexTypeEditor.exe”

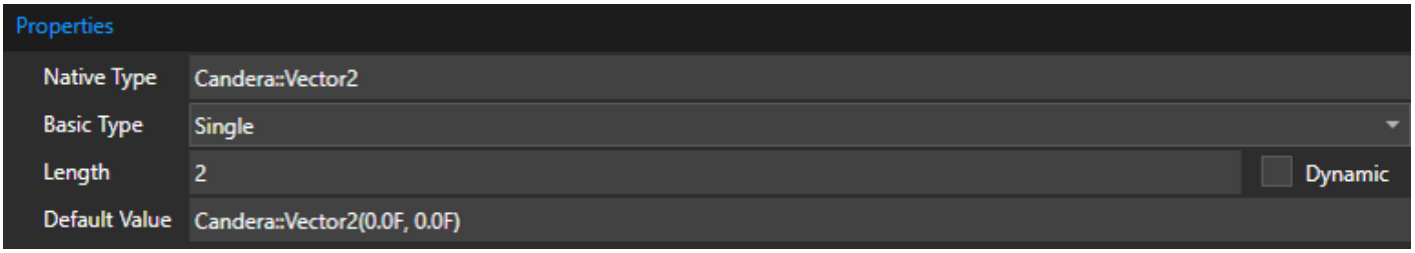
If called with a file as first argument the editor will try to open the specified file.

Complex-Types



On the left side, Complex-Types can be added/removed and reordered using the respective buttons.

Properties:



Property Name	Description
Native Type	C/C++ name of the native type
Basic Type	Basic type used to represent the Complex-Type. This type will be used when transferring data via the Interface.
Length	Length of the Complex-Type based on the Basic-Type. To use to a dynamic length (e.g. an array) the “Dynamic” checkbox can be used.
Default Value	Default value used for this Complex-Type.