

```
#pragma once
```

```
#include <algorithm> // std::min
```

```
/******
```

```
// SCString
```

```
class SCString
```

```
{
```

```
    friend class c_Graph;
```

```
private:
```

//When this is not a null pointer and it is not equal to m_DefaultString, then when m_OwnString is 0 it is pointing to a string allocated elsewhere. When m_OwnString is 1, it is pointing to a string which is allocated by an instance of this object. This class is used only by ACSIL. This design of referencing a string allocated elsewhere, needs to be removed and when we need to access those strings, we need to use a function instead and not this class because this design is inherently unsafe.

```
    char* m_String;
    int m_OwnString;
    int m_IsModified;
    const char* m_DefaultString;
```

```
public:
```

```
    SCString();
    SCString(int DummyValue); // this is needed by the c_ArrayWrapper class
    SCString(const SCString& Str); // Copy constructor
    SCString(const char* SourceString);
    SCString(const char* SourceString, int Length);
    SCString(const char Character);
    ~SCString();
```

```
void Initialize();
```

```
void Clear();
```

```
bool operator == (const SCString& Rhs) const;
bool operator == (const char* Rhs) const;
bool operator != (const SCString& Rhs) const;
bool operator != (const char* Rhs) const;
bool operator < (const SCString& Rhs) const;
bool operator < (const char* Rhs) const;
SCString& operator = (const SCString& Rhs);
SCString& operator = (const char* String);
SCString& operator += (const SCString& Rhs);
SCString operator + (const SCString& Rhs) const;
```

```
operator const char* () const;
```

```
SCString& Format(const char* String, ...);
SCString& AppendFormat(const char* String, ...);
```

```
int Compare(const char* StringToCompare, int NumChars = 0) const;
int CompareNoCase(const char* StringToCompare, int NumChars = 0) const;
int CompareNoCase(const SCString& StringToCompare, int NumChars = 0) const;
```

```
int IsModified() const;
int IsEmpty() const;
```

```
const char* GetChars() const;
int GetLength() const;
```

```
int IndexOf(char Delimiter, int StartIndex = 0) const;
int LastIndexOf(char SearchCharacter, int StartIndex) const;
```

```

SCString GetSubString(int SubStringLength, int StartIndex = 0) const;

void ParseLines(std::vector<SCString> &Lines);
void ParseLineItemsAsFloats(std::vector<float> &FloatValues);

int Tokenize(const char* Delim, std::vector<char*> &Tokens);

SCString& Append(const SCString& Rhs);

SCString Left(int Count) const;
SCString Right(int Count) const;

private:
    // The maximum allowed length of a string is the maximum value that
    // can be stored in an int, -1 so that the buffer size which has to
    // include the null terminating character can also be stored in an
    // int.
    static const int MAX_LENGTH = INT_MAX - 1;

private:
    static int TerminatedStringLength(const char* String);

    static char* StringNew(const int NumBytes);
    static void StringDelete(char* String);

    SCString& Copy(const char* String, int StringLength);

    void InternalAppendFormat
        ( const char* String
          , const va_list& ArgumentList
          );

    // For Sierra Chart internal use only
    void InternalSetStatic(const char* String);
};

/*=====*/
inline SCString::SCString()
{
    m_DefaultString = "";
    m_String = (char*)m_DefaultString; //NULL
    m_OwnString = 0;
    m_IsModified = 0;
}

/*=====
This constructor is needed by the c_ArrayWrapper class.
=====*/
inline SCString::SCString(int DummyValue)
{
    m_DefaultString = "";
    m_String = (char*)m_DefaultString; //NULL
    m_OwnString = 0;
    m_IsModified = 0;
}

/*=====*/
inline SCString::SCString(const SCString& Str)
{
    m_DefaultString = "";
    m_String = (char*)m_DefaultString; //NULL
    m_OwnString = 0;
    m_IsModified = Str.m_IsModified;
}

```

```

const int StringLength = TerminatedStringLength(Str.m_String);

if (StringLength != 0)
{
    const int BufferSize = StringLength + 1;

    m_String = StringNew(BufferSize);
    if (m_String != NULL)
    {
        #if __STDC__WANT_SECURE_LIB__
            strcpy_s(m_String, BufferSize, Str.m_String);
        #else
            strcpy(m_String, Str.m_String);
        #endif
        m_OwnString = 1;
    }
    else
    {
        m_String = (char*)m_DefaultString; //NULL
    }
}
else
{
    m_String = (char*)m_DefaultString; //NULL
}
}

/*=====*/
inline SCString::SCString(const char* SourceString)
{
    m_DefaultString = "";
    m_String = (char*)m_DefaultString; //NULL
    m_OwnString = 0;
    m_IsModified = 0;

    const int StringLength = TerminatedStringLength(SourceString);

    if (StringLength != 0)
    {
        const int BufferSize = StringLength + 1;

        m_String = StringNew(BufferSize);
        if (m_String != NULL)
        {
            #if __STDC__WANT_SECURE_LIB__
                strncpy_s(m_String, BufferSize, SourceString, StringLength);
            #else
                strncpy(m_String, SourceString, StringLength);
            #endif
            m_OwnString = 1;
            m_IsModified = 1;
        }
        else
        {
            m_String = (char*)m_DefaultString; //NULL
        }
    }
    else
    {
        m_String = (char*)m_DefaultString; //NULL
    }
}

/*=====*/
inline SCString::SCString(const char* SourceString, int Length)
{

```

```

m_DefaultString = "";
m_String = (char*)m_DefaultString; //NULL
m_OwnString= 0;
m_IsModified= 0;

int StringLength = 0;
if (SourceString != NULL)
{
    if (Length >= 0)
    {
        // Note: if `min` is defined as a macro, that prevents us from
        // using `std::min`. But if `min` is not defined, we need to
        // specify the `std` namespace.
#ifdef min
        StringLength = min(Length, MAX_LENGTH);
#else
        StringLength = std::min(Length, MAX_LENGTH);
#endif
    }
    else
        StringLength = TerminatedStringLength(SourceString);
}

if (StringLength != 0)
{
    const int BufferSize = StringLength + 1;

    m_String = StringNew(BufferSize);
    if (m_String != NULL)
    {
#ifdef __STDC_WANT_SECURE_LIB__
        strncpy_s(m_String, BufferSize, SourceString, StringLength);
#else
        strncpy(m_String, SourceString, StringLength);
#endif
        m_OwnString = 1;
        m_IsModified = 1;
    }
    else
    {
        m_String = (char*)m_DefaultString; //NULL
    }
}
else
    m_String = (char*)m_DefaultString; //NULL
}

/*=====*/
inline SCString::SCString(const char Character)
{
    m_DefaultString = "";
    m_String = (char*)m_DefaultString; //NULL
    m_OwnString = 0;
    m_IsModified = 0;

    const int StringLength = 1;
    const int BufferSize = StringLength + 1;

    m_String = StringNew(BufferSize);
    if (m_String != NULL)
    {
        m_String[0] = Character;
        m_String[StringLength] = '\0';
        m_OwnString = 1;
    }
}

```

```

        m_IsModified = 1;
    }
    else
        m_String = (char*)m_DefaultString; //NULL
}

/*=====*/
inline SCString::~SCString()
{
    if (m_OwnString && m_String != NULL)
    {
        StringDelete(m_String);
        m_String = nullptr;
        m_OwnString = 0;
        m_IsModified = 0;
    }
}

/*=====*/
inline void SCString::Initialize()
{
    if (m_OwnString && m_String != NULL)
        StringDelete(m_String);

    m_String = (char*)m_DefaultString; //NULL
    m_OwnString = 0;
    m_IsModified = 0;
}

/*=====*/
inline void SCString::Clear()
{
    Initialize();
}

/*=====*/
inline bool SCString::operator == (const SCString& RhS) const
{
    return (Compare(RhS.GetChars()) == 0);
}

/*=====*/
inline bool SCString::operator == (const char* RhS) const
{
    return (Compare(RhS) == 0);
}

/*=====*/
inline bool SCString::operator != (const SCString& RhS) const
{
    return (Compare(RhS.GetChars()) != 0);
}

/*=====*/
inline bool SCString::operator != (const char* RhS) const
{
    return (Compare(RhS) != 0);
}

/*=====*/
inline bool SCString::operator < (const SCString& RhS) const
{
    return Compare(RhS.GetChars()) < 0;
}

```

```

}

/*=====*/
inline bool SCString::operator < (const char* RhS) const
{
    return Compare(Rhs) < 0;
}

/*=====*/
inline SCString& SCString::operator = (const SCString& RhS)
{
    if (&RhS == this)
        return *this;

    if (RhS.IsEmpty() && IsEmpty() )
        return *this;

    if (RhS.m_String != nullptr)
        Copy(RhS.m_String, TerminatedStringLength(RhS.m_String));
    else
        Initialize();

    m_IsModified = 1;

    return *this;
}

/*=====*/
inline SCString& SCString::operator = (const char* String)
{
    if ((String == NULL || String[0] == '\0') && IsEmpty())
        return *this; //nothing to do

    if (String != nullptr)
        Copy(String, TerminatedStringLength(String));
    else
        Initialize();

    m_IsModified = 1;

    return *this;
}

/*=====*/
inline SCString& SCString::operator += (const SCString& RhS)
{
    return Append(Rhs);
}

/*=====*/
inline SCString SCString::operator + (const SCString& RhS) const
{
    SCString Result(*this);
    Result += RhS;
    return Result;
}

/*=====*/
inline SCString::operator const char* () const
{
    if (m_String != nullptr)
        return m_String;
    else
        return "";
}

```

```

/*=====*/
inline SCString& SCString::Format(const char* String, ...)
{
    Initialize();

    va_list ArgumentList;
    va_start(ArgumentList, String);

    InternalAppendFormat(String, ArgumentList);

    va_end(ArgumentList);

    return *this;
}

/*=====*/
inline SCString& SCString::AppendFormat(const char* String, ...)
{
    va_list ArgumentList;
    va_start(ArgumentList, String);

    InternalAppendFormat(String, ArgumentList);

    va_end(ArgumentList);

    return *this;
}

/*=====*/
inline int SCString::Compare(const char* StringToCompare, int NumChars) const
{
    const char* InternalString= "";
    if (m_String != nullptr)
        InternalString=m_String;

    if (StringToCompare != NULL)
    {
        if (NumChars == 0)
            return strcmp(StringToCompare, InternalString);
        else
            return strncmp(StringToCompare, InternalString, NumChars);
    }
    else
        return 0;
}

/*=====*/
inline int SCString::CompareNoCase(const char* StringToCompare, int NumChars) const
{
    const char* InternalString= "";
    if (m_String != nullptr)
        InternalString=m_String;

    #if _MSC_VER >= 1400
    if (NumChars == 0)
        return _stricmp(StringToCompare, InternalString);
    else
        return _strnicmp(StringToCompare, InternalString, NumChars);
    #else
    if (NumChars == 0)
        return strcmp(StringToCompare, InternalString);
    else
        return strnicmp(StringToCompare, InternalString, NumChars);
    #endif
}

```

```

#endif
}

/*=====*/
inline int SCString::CompareNoCase(const SCString& StringToCompare, int NumChars) const
{
    return CompareNoCase(StringToCompare.GetChars(),NumChars);
}

/*=====*/
inline int SCString::IsModified() const
{
    return m_IsModified;
}

/*=====*/
inline int SCString::IsEmpty() const
{
    if (m_String == NULL)
        return 1;

    return (m_String[0] == '\0') ? 1 : 0;
}

/*=====*/
inline const char* SCString::GetChars() const
{
    if (m_String != nullptr)
        return m_String;
    else
        return "";
}

/*=====*/
inline int SCString::GetLength() const
{
    return TerminatedStringLength(m_String);
}

/*=====*/
inline int SCString::IndexOf(char Delimiter, int StartIndex) const
{
    if (m_String == NULL)
        return -1;

    for (int i = StartIndex; i < GetLength(); i++)
    {
        if (m_String[i] == Delimiter)
            return i;
    }
    return -1;
}

/*=====*/
inline int SCString::LastIndexOf(char SearchCharacter, int StartIndex) const
{
    if (m_String == NULL)
        return -1;

    const int Length = GetLength();

    if (StartIndex >= Length)
        StartIndex = Length - 1;

    for (int CharacterIndex = StartIndex; CharacterIndex >= 0; CharacterIndex--)

```



```

{
    if (m_String[CharacterIndex] == SearchCharacter)
        return CharacterIndex;
}

return -1;
}

/*=====*/
inline SCString SCString::GetSubString(int SubstringLength, int StartIndex) const
{
    if (m_String == NULL)
        return SCString();

    const int Length = GetLength();
    if (StartIndex >= Length)
        return SCString();

    if (SubstringLength > Length - StartIndex)
        SubstringLength = Length - StartIndex;

    SCString Result;
    Result.Copy(m_String + StartIndex, SubstringLength);

    return Result;
}

/*=====*/
inline void SCString::ParseLines(std::vector<SCString>& Lines)
{
    int StartIndex = 0;
    while(true)
    {
        int NewlineIndexPosition = IndexOf('\n', StartIndex);

        if (NewlineIndexPosition == -1)
            NewlineIndexPosition = GetLength();

        SCString Line;
        Line = GetSubString(NewlineIndexPosition - StartIndex, StartIndex);
        Lines.push_back(Line);
        StartIndex = NewlineIndexPosition + 1;

        if(StartIndex >= GetLength())
            break;
    }
}

/*=====*/
inline void SCString::ParseLineItemsAsFloats(std::vector<float>& FloatValues)
{
    char FieldsDelimiter = ',';

    int find_delim = IndexOf(FieldsDelimiter);
    int pos = 0;

    int ArraySize = 1000;
    int CurrentItem = 0;

    while (find_delim > 0 && CurrentItem < ArraySize)
    {
        SCString NextStr = GetSubString(find_delim - pos, pos);

```

```

FloatValues.push_back(static_cast<float>(atof(NextStr)));
pos = find_delim + 1;
find_delim = IndexOf(FieldsDelimiter,find_delim + 1);

CurrentItem++;
}
}

/*=====*/
inline int SCString::Tokenize(const char* Delim, std::vector<char*>& Tokens)
{
    Tokens.erase(Tokens.begin(), Tokens.end());

    if (m_String == NULL)
        return 0;

    // The object must own the string for this method because this method
    // modifies the string.
    if (!m_OwnString)
        Copy(m_String, TerminatedStringLength(m_String));

    Tokens.push_back(m_String);

    if (Delim == NULL)
        return 1;

    int StrLen = GetLength();
    int DelimLen = TerminatedStringLength(Delim);

    if (DelimLen == 0)
        return 1; // Not worth splitting every character

    for (int a = 0; a < StrLen; a++)
    {
        // Look for the delimiter
        bool DelimFound = true;
        for (int b = 0; b < DelimLen; b++)
        {
            if (a + b >= StrLen)
            {
                DelimFound = false;
                break;
            }

            if (m_String[a + b] != Delim[b])
            {
                DelimFound = false;
                break;
            }
        }

        if (DelimFound)
        {
            m_String[a] = '\0';
            Tokens.push_back(m_String + a + DelimLen);
            a += DelimLen - 1;
        }
    }

    return static_cast<int>(Tokens.size());
}

/*=====*/
inline SCString& SCString::Append(const SCString& Rhs)
{

```

```

int RightStringLength = Rhs.GetLength();
if (RightStringLength == 0)
    return *this;

const int LeftStringLength = GetLength();

// Make sure the new string length does not exceed MAX_LENGTH.
if (RightStringLength > MAX_LENGTH - LeftStringLength)
    RightStringLength = MAX_LENGTH - LeftStringLength;

const int NewStringLength = LeftStringLength + RightStringLength;

char* NewString = StringNew(NewStringLength + 1);
if (NewString != nullptr)
{
    const char* InternalString = "";
    if (m_String != nullptr)
        InternalString = m_String;

    #if __STDC__ WANT_SECURE_LIB__
    strncpy_s(NewString, NewStringLength + 1, InternalString, LeftStringLength);
    strncpy_s(NewString + LeftStringLength, NewStringLength + 1 - LeftStringLength, Rhs.m_String, RightStringLength);
    #else
    strncpy(NewString, InternalString, LeftStringLength);
    strncpy(NewString + LeftStringLength, Rhs.m_String, RightStringLength);
    #endif
    NewString[NewStringLength] = '\0';

    Initialize();
    m_String = NewString;
    m_OwnString = 1;
}

m_IsModified = 1;

return *this;
}

/*=====*/
inline SCString SCString::Left(int Count) const
{
    const int StringLength = GetLength();

    if (Count >= StringLength)
        return *this;
    else if (Count > 0)
        return SCString(m_String, Count);
    else if (Count < 0)
    {
        int TargetLength = StringLength + Count;
        if (TargetLength <= 0)
            return SCString();
        else
            return SCString(m_String, TargetLength);
    }
    else
        return SCString();
}

/*=====*/
inline SCString SCString::Right(int Count) const
{
    int StringLength = GetLength();

    if (Count >= StringLength)

```

```

        return *this;
    else if (Count > 0)
        return SCString(m_String + (StringLength - Count), Count);
    else if (Count < 0)
    {
        int TargetLength = StringLength + Count;
        if (TargetLength <= 0)
            return SCString();
        else
            return SCString(m_String + (StringLength - TargetLength), TargetLength);
    }
    else
        return SCString();
}

```

```

/*=====
Static
-----*/

```

```

inline int SCString::TerminatedStringLength(const char* String)
{
    if (String == NULL)
        return 0;

    const size_t StringLength = strlen(String);

    if (StringLength > MAX_LENGTH)
        return MAX_LENGTH;

    return static_cast<int>(StringLength);
}

```

```

/*=====
Static
-----*/

```

```

inline char* SCString::StringNew(const int NumBytes)
{
    if (NumBytes <= 0)
        return NULL;

```

//The reason that the heap allocation functions in the operating system API are being used rather than the "new" operator is because a custom study DLL which is using the s_sc structure which contains SCString members can be unloaded during debugging and code modification. These pointers are still remembered within an instance of s_sc which is a member of c_Graph which resides within the SierraChart.exe module. An exception will occur when using delete after the DLL has been freed and reloaded. Whereas using the operating system heap allocation functions, this exception is avoided.

```

    char* p_Block = (char*)HeapAlloc(GetProcessHeap(), 0, NumBytes);

    if (p_Block != NULL)
        memset(p_Block, 0, NumBytes);

    return p_Block;
}

```

```

/*=====
Static
-----*/

```

```

inline void SCString::StringDelete(char* String)
{
    if (String != NULL)
        HeapFree(GetProcessHeap(), 0, String);
}

```

```

/*=====*/
inline SCString& SCString::Copy(const char* String, int StringLength)
{

```

```

if (String == NULL)
    return *this;

Initialize();

if (StringLength != 0)
{
    if (StringLength > MAX_LENGTH)
        StringLength = MAX_LENGTH;

    const int BufferSize = StringLength + 1;

    m_String = StringNew(BufferSize);

    if (m_String == NULL)
        m_String = (char*)m_DefaultString;
    else
    {
        #if __STDC__ WANT_SECURE_LIB__
            strncpy_s(m_String, BufferSize, String, StringLength);
        #else
            strncpy(m_String, String, StringLength);
        #endif
        m_String[StringLength] = '\0';
        m_OwnString = 1;
    }
}

m_IsModified = 1;

return *this;
}

/*=====*/
inline void SCString::InternalAppendFormat
( const char* FormatString
, const va_list& ArgumentList
)
{
    if (FormatString == NULL)
        return;

    const int PriorLength = TerminatedStringLength(m_String);

    const int TargetLength = _vsnprintf(FormatString, ArgumentList);

    if (TargetLength < 0 || TargetLength > MAX_LENGTH - PriorLength)
        return;

    char* NewStringBuffer = StringNew(PriorLength + TargetLength + 1);
    if (NewStringBuffer == NULL)
        return;

    // Copy the current string to the new buffer.
    if (PriorLength > 0)
        memcpy(NewStringBuffer, m_String, PriorLength);

    #if __STDC__ WANT_SECURE_LIB__
        vsprintf_s
        ( NewStringBuffer + PriorLength
        , TargetLength + 1
        , FormatString
        , ArgumentList
        );
    #else

```

```

vsprintf(NewStringBuffer + PriorLength, FormatString, ArgumentList);
#endif

Clear();

m_String = NewStringBuffer;
m_OwnString = 1;
m_IsModified = 1;
}

/*=====
   For Sierra Chart internal use only.
-----*/
inline void SCString::InternalSetStatic(const char* String)
{
    Initialize();

    m_String = const_cast<char*>(String);
    m_OwnString = 0;
    m_IsModified = 0;
}

/*=====*/

```