Aufgabe M3.1

Ich habe versucht, die Funktionalität als Baustein in Klassen zu verpacken. Mein Programm enthält daher keine Routinen zur Ein- / Ausgabe, falls man zu Testzwecken die Abläufe verfolgen will, empfehle ich eine Debug-Sitzung.

Die Kernroutine zur Entfernung der überflüssigen Leerzeichen durchläuft den kompletten String. Jedes Auftreten eines Leerzeichens hat zur Folge, dass in einer Schleife alle direkt darauf folgenden Leerzeichen entfernt werden.

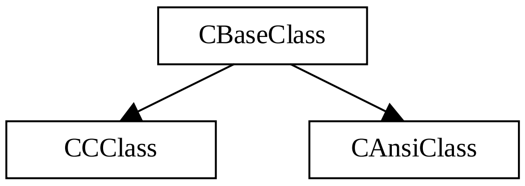
```
// get length
int nSize = strSource.GetLength();

// take a look at each character
for (int nLoop = 0; nLoop < nSize; nLoop++)
{
        // space found ?
        if (strSource.GetAt(nLoop) == ' ')
                // remove all trailing spaces
                while (strSource.GetAt(nLoop+1) == ' ')
                {
                        strSource.DeleteAt(nLoop+1);
                        // decrease size counter
                        nSize--;
                }
}
```

Als Resultat erhält man einen String, der keine direkt aufeinanderfolgende Leerzeichen besitzt. Die im Programm implementierte Version entfernt zusätzlich noch eventuell auftretende den String anführende bzw. abschließende Leerzeichen.

Um die Eigenschaften eines Interfaces unter Java nachzuahmen, definiere ich eine Klasse `CBaseString`, die nur rein virtuelle Funktionen bereitstellt. Jede davon abgeleitete Klasse ist somit gezwungen, diese Funktionalität garantiert bereitzustellen.

Ein C-String bzw. ein ANSI(-Pascal)-String kann durch `CCString` bzw. `CAnsiString` dargestellt werden. Die Klassenhierachie sieht dementsprechend sehr einfach aus:



Die implementierten Funktionen sind:

```
// clear string
virtual void Empty();
// get length of the string, 0 on error
virtual int  GetLength();
// get a character at a specified position, 0 on error
virtual char GetAt(int nIndex);
// set a character at a specified position
virtual void SetAt(int nIndex, char cCharacter);

// append a character
virtual void Append(char cCharacter);
// delete a specified character
virtual void DeleteAt(int nIndex);
```

Zusätzlich enthalten `CCString` und `CansiString` einen Konstruktor, der eine Anfangsbelegung erlaubt und einen Destruktor, der benutzten Speicher wieder ordnungsgemäß freigibt.

```
// construct a new string, set initial content and block size
CAnsiString(char* strInitial = NULL, int nGrowBy = 10);
```

```
        // destroy it
        virtual ~CAnsiString();
```

Die einzelnen Code-Zeilen bespreche ich nicht weiter, da ich den Quelltext sehr ausführlich (auf Englisch) kommentiert habe.

String.h:

```
///////////////////////////////////////////////////////////
// Softwarebauelemente I, Aufgabe M3.1
//
// author:            Stephan Brumme
// last changes:      October 26, 2000


// avoid multiple compiling
#if !defined(STRING_H)
#define STRING_H
#pragma once


// include to use string copy operations etc.
#include <string.h>


// the base class defines an standard interface for inherited class
// all functions are pure virtual
class CBaseString
{
public:
        // clear string
        virtual void Empty() =0;
        // get length of the string, 0 on error
        virtual int  GetLength() =0;
        // get a character at a specified position, 0 on error
        virtual char GetAt(int nIndex) =0;
        // set a character at a specified position
        virtual void SetAt(int nIndex, char cCharacter) =0;

        // append a character
        virtual void Append(char cCharacter) =0;
        // delete a specified character
        virtual void DeleteAt(int nIndex) =0;

protected:
        // stores the string, structure depends upon inherited class
        char* m_pStorage;
};



// class represents a C-style string
class CCString : public CBaseString
{
public:
        // construct a new string, set initial content and block size
        CCString(char* strInitial = NULL, int nGrowBy = 10);
        // destroy it
        virtual ~CCString();

        // clear string
        virtual void Empty();
        // get length of the string, 0 on error
        virtual int  GetLength();
        // get a character at a specified position, 0 on error
        virtual char GetAt(int nIndex);
        // set a character at a specified position
        virtual void SetAt(int nIndex, char cCharacter);

        // append a character
        virtual void Append(char cCharacter);
        // delete a specified character
        virtual void DeleteAt(int nIndex);

protected:
        // allocated memory size
        int m_nStorageSpace;
        // avoid re-allocating memory every time by using pre-allocation
        int m_nGrowBy;
};



// class represents a ANSI-style string (just like Pascal)
class CAnsiString : public CBaseString
{
public:
        // construct a new string, set initial content and block size
        CAnsiString(char* strInitial = NULL, int nGrowBy = 10);
        // destroy it
        virtual ~CAnsiString();

        // clear string
        virtual void Empty();
        // get length of the string, 0 on error
        virtual int  GetLength();
        // get a character at a specified position, 0 on error
        virtual char GetAt(int nIndex);
        // set a character at a specified position
        virtual void SetAt(int nIndex, char cCharacter);

        // append a character
        virtual void Append(char cCharacter);
        // delete a specified character
        virtual void DeleteAt(int nIndex);

protected:
        // allocated memory size
        int m_nStorageSpace;
        // avoid re-allocating memory every time by using pre-allocation
        int m_nGrowBy;
};

#endif // !defined(STRING_H)
```

String.cpp:

```
///////////////////////////////////////////////////////////
// Softwarebauelemente I, Aufgabe M3.1
//
// author:            Stephan Brumme
// last changes:      October 26, 2000


#include "String.h"


///////////////////////////////////////////////////////////
// CCString

// construct a new string, set initial content and block size
CCString::CCString(char* strInitial, int nGrowBy)
{
        // set default values
        m_pStorage = NULL;
        m_nStorageSpace = 0;
        m_nGrowBy = nGrowBy;

        // if no initial string given then we are done
        if (strInitial == NULL)
                return;

        // we have a initial string
        // get required memory size
        m_nStorageSpace = strlen(strInitial)+1;
        // allocate some memory
        m_pStorage = new char[m_nStorageSpace];
        // store the string
        strcpy(m_pStorage, strInitial);
}


// destroy it
CCString::~CCString()
{
        // delete any content
        Empty();
}


// clear string
void CCString::Empty()
{
        // already empty ?
        if (m_pStorage == NULL)
               return;

        // free memory
        delete(m_pStorage);

        // reset to defaults values
        m_pStorage = NULL;
        m_nStorageSpace = 0;
}


// get length of the string, 0 on error
int CCString::GetLength()
{
        // is string empty ? => error
        if (m_pStorage == NULL)
               return 0;

        // use C-function to get string size
        return strlen(m_pStorage);
}


// get a character at a specified position, 0 on error
char CCString::GetAt(int nIndex)
{
        // is string too short ? => error
        if (nIndex > GetLength())
               return 0;

        // return the character
        return m_pStorage[nIndex];
}


// set a character at a specified position
void CCString::SetAt(int nIndex, char cCharacter)
{
        // is string too short ? => error
        if (nIndex > GetLength())
               return;

        // set character
        m_pStorage[nIndex] = cCharacter;
}


// append a character
void CCString::Append(char cCharacter)
{
        // get length
        int nSize = GetLength();

        // do we need more memory ?
        if (nSize+2 > m_nStorageSpace)
        {
               // calculate new memory requirements
               m_nStorageSpace += m_nGrowBy;
               // allocate new memory
               char* pNewStorage = new char[m_nStorageSpace];

               // copy the stored string
               strcpy(pNewStorage, m_pStorage);
               // free old storage memory
               delete(m_pStorage);
               // redirect to new storage memory
               m_pStorage = pNewStorage;
        }

        // set character
        m_pStorage[nSize]   = cCharacter;
        // add terminating zero
        m_pStorage[nSize+1] = 0;
}


// delete a specified character
void CCString::DeleteAt(int nIndex)
{
        // get length
        int nSize = GetLength();

        // is string too short ? => error
        if (nIndex > nSize)
               return;

        // copy each trailing character
        // including the final zero
        while (nIndex < nSize)
        {
               char cMove = GetAt(nIndex+1);
               SetAt(nIndex, cMove);

               nIndex++;
        }
}



///////////////////////////////////////////////////////////
// CAnsiString

// construct a new string, set initial content and block size
CAnsiString::CAnsiString(char* strInitial, int nGrowBy)
{
        // set default values
        m_pStorage = NULL;
        m_nStorageSpace = 0;
        m_nGrowBy = nGrowBy;

        // if no initial string given then we are done
        if (strInitial == NULL)
               return;

        // we have a initial string
        // get length
        int nLength = strlen(strInitial);
        // get required memory size
        m_nStorageSpace = nLength + 1;

        // allocate some memory
        m_pStorage = new char[m_nStorageSpace];
        // store the string
        memcpy(m_pStorage+1, strInitial, nLength);

        // store length
        m_pStorage[0] = nLength;
}


// destroy it
```

```
CAnsiString::~CAnsiString()
{
        // delete any content
        Empty();
}


// clear string
void CAnsiString::Empty()
{
        // already empty ?
        if (m_pStorage == NULL)
                return;

        // free memory
        delete(m_pStorage);

        // reset to defaults values
        m_pStorage = NULL;
        m_nStorageSpace = 0;
}


// get length of the string, 0 on error
int CAnsiString::GetLength()
{
        // is string empty ? => error
        if (m_pStorage == NULL)
                return 0;

        // get string size from leading byte
        return m_pStorage[0];
}


// get a character at a specified position, 0 on error
char CAnsiString::GetAt(int nIndex)
{
        // is string too short ? => error
        if (nIndex > GetLength())
                return 0;

        // return the character (care for 1 byte offset due to stored size byte)
        return m_pStorage[nIndex+1];
}


// set a character at a specified position
void CAnsiString::SetAt(int nIndex, char cCharacter)
{
        // is string too short ? => error
        if (nIndex > GetLength())
                return;

        // set character (care for 1 byte offset due to stored size byte)
        m_pStorage[nIndex+1] = cCharacter;
}


// append a character
void CAnsiString::Append(char cCharacter)
{
        // get length
        int nSize = GetLength();

        // do we need more memory ?
        if (nSize+2 > m_nStorageSpace)
        {
                // calculate new memory requirements
                m_nStorageSpace += m_nGrowBy;
                // allocate new memory
                char* pNewStorage = new char[m_nStorageSpace];

                // copy the stored string
                memcpy(pNewStorage, m_pStorage, nSize+1);
```

```
		// free old storage memory
		delete(m_pStorage);
		// redirect to new storage memory
		m_pStorage = pNewStorage;
	}

	// set character (don't forget the magic 1 byte offset)
	m_pStorage[nSize+1] = cCharacter;
	// increase size byte
	m_pStorage[0]++;
}


// delete a specified character
void CAnsiString::DeleteAt(int nIndex)
{
	// get length
	int nSize = GetLength();

	// is string too short ? => error
	if (nIndex > nSize)
		return;

	// copy each trailing character
	// care for 1 byte story
	while (nIndex+1 < nSize)
	{
		char cMove = GetAt(nIndex+1);
		SetAt(nIndex, cMove);

		nIndex++;
	}

	// decrease size byte
	m_pStorage[0]--;
}
```

M03_1.cpp:

```
////////////////////////////////////////////////////////////
// Softwarebauelemente I, Aufgabe M3.1
//
// author:			Stephan Brumme
// last changes:	October 26, 2000



// we use classes CAnsiString, CCString and their base class CBaseString
#include "String.h"


// remove unnecessary spaces from a string
// function is able to handle all strings classes inherited from CBaseString
void RemoveSpaces(CBaseString &strSource)
{
	// get length
	int nSize = strSource.GetLength();

	// take a look at each character
	for (int nLoop = 0; nLoop < nSize; nLoop++)
	{
		// space found ?
		if (strSource.GetAt(nLoop) == ' ')
			// remove all trailing spaces
			while (strSource.GetAt(nLoop+1) == ' ')
			{
				strSource.DeleteAt(nLoop+1);
				// decrease size counter
				nSize--;
			}
	}

	// remove leading space
	if (strSource.GetAt(0) == ' ')
```

```
                strSource.DeleteAt(0);

        // remove trailing space
        nSize = strSource.GetLength();
        if (strSource.GetAt(nSize-1) == ' ')
                strSource.DeleteAt(nSize-1);
}



void main()
{
        // some test code
        CCString MyCString("  Ha   llo !   ");
        CAnsiString MyAnsiString("  Ha   llo !   ");

        RemoveSpaces(MyCString);
        RemoveSpaces(MyAnsiString);
}
```