Document :: AnsiString
Version 4
Author :: Jon Jenkinson, (With much credit to Kent Reisdorph for his advice and clarification)

From "The Bits" Website
http://www.cbuilder.dthomas.co.uk
email : jon.jenkinson@mcmail.com

Version Information

Version 4	*: Converted to PDF format by Kris Erickson, edited document...*
Version 3	*: Updated function details supplied by Kris Erickson*
Version 2	*: Advice and clarification supplied by Kent Reisdorph*
Version 1	*: Original By Jon Jenkinson*

AnsiString

Ah, AnsiString, much loved by Pascal programmers, beginning to be hated by C++ Builder programmers. This really is a shame as AnsiString has benefits to those of us from a C background, and whilst not perfect, this is usually due to the lack of decent documentation.

A bit Of History

AnsiString is a class, and as such has many member functions which make it far easier to use strings than the old fashioned way of *"C"*. I say old fashioned because I am a C programmer using BCB as his stepping stone to C++. This has led to a couple of misunderstandings by me, misunderstandings which I am sure I am not alone in making. Some clarification,

C has no data type for a string. In C we used a simple array of characters terminated by a NULL,

i.e.

char a; is a single variable capable of holding a single character, (a = 'a');
char a[255], is an array of single characters, in the same way as any array.

In C we can assign a string to an array because this is, if you will, a work around by the compiler vendors to make up for the lack of a genuine string data type, thus,

```
char *a = "A char";
```

(editors note: Actually a char* string is a pointer that points to the data segment of the program in memory, thus if you change a in the following example, you will be changing the program, possibly modifying other data, so you should always make char* constants, or you could end up writing self modifying code).

is the same as us having to type (editors note: actually it is not since the following places allocates space for a on the stack, whereas the above example stores the information in the programs data segment...)

```
char a[5];
a[0]= 'A';
a[1] = ' ';
a[2] = 'c';
a[3] = 'h';
a[4] = 'a';
a[5] = 'r';
```

Painful eh ? The fact that C is a language which works well with pointers allows us to do things much more easily, and you would normally see the above example in C in the following form,

```
char *a;                                    // a pointer to a char

a = new char[255];             // allocation of memory to hold the array;
strcpy(a,"This is a string"); // assign a,  string of chars,  to the
                                      // memory location pointed to be a.
```

This stores the *string of chars* at the in/ memory, and adds the null terminator, '\0'. When we use C functions to read strings, strlen, for instance, it effectively reads through the array of characters until it reaches the null terminator.,
e.g.

```
int strlen(char *source)
{
        int x = 0;
        while(source[x++]);
        return x--;
}
```

As you can see, the now standard C libraries have taken up a lot of the work for us when it comes to using strings, but there are many mistakes which come from a misunderstanding of the basic problem for C, there is no string type.

C does not check the bounds of your string, i.e. the following will compile, and create havoc at run time.

```
char *string;
```

```
string = new char [5];                  //allocate space for array
strcpy(string,"This is a string");    //will compile,  but whoooops!
```

The reason that problems will arise is that you've asked the compiler to allocate space for 4 characters, (**four plus the null terminator**), and then assigned a *string of chars* without a null terminator. Consider the following,

```
#include <stdio.h>
#include <string.h>

void main(void) {

  char *astring, *anotherString;

  astring = new char[5];
  anotherString = new char [255];

  strcpy(astring,"this is a string");
  strcpy(anotherString,"THIS IS ANOTHER STRING");

  printf("astring: %s\n",astring);
  printf("another string: %s\n",anotherString);

}
```

As before, it will compile without complaint, but at runtime, the memory location pointed to by astring will not contain the *string of chars* you expect,

[index of array]
astring = [t][h][i][s][ ]

note the lack of a null terminator.

In fact, in string terms, the memory pointed to by astring contains the following,
astring = [t][h][i][s][ ][i][T][H][I][S][ ][I][S][ ][A][N][O][T][H][E][R][ ][S][T][R][I][N][G][\0]

(editor's note: This is not necessarily the case, it depends where anotherString was placed into the symbol table.)

This shows one of the commonest *bugs* in C programming. The length of astring returned by strlen would be 28 and not the 5 we would expect.

<u>So What Has All This Got To Do With AnsiString ?</u>

A lot. With the advent of C++ classes were designed to handle strings in a more manageable fashion than under C. The essence of this is that classes took most of the functions we had all come to know and love, and made them into member functions of the class. (strcat becomes an overload of the +, strcpy becomes an overload of the =, strlen becomes the length() function……and a major one, dynamic string memory handling, that is no malloc no silly errors).

There is no defacto standard to C++ string handling, as most vendors produced their own. When Borland ported the Delphi(Pascal) VCL to C++ it had a problem. Delphi has built in string data types, PChar, and Long String being the two we are interested in. PChar, is, (guess what), a char pointer, and Long String is a string type limited only by available memory. (Delphi does provide other string types, but they are of no interest to us.)

Now in C, a PChar is effectively a char *, but there was no equivalent of the Delphi Long String that Borland used in the make up of the VCL. **Thus they defined AnsiString, *a string class*, not a data type**.

AnsiString has many advantages to us as programmers. The two major ones are that it is at the heart of the VCL, with most text handling in the VCL being done in the AnsiString class. The second, and most C++ programmers already know this one, is that the allocation of memory is dynamic, for us C guys, read no need to worry about bounds checking)

As a class however, AnsiString does have a downside, it is non-portable and slow, (not slow enough to notice however.) Below is a list of plus' for both a standard char array and AnsiString.

Char array(char *)	AnsiString
speed	Dynamic allocation, no bounds checking
portability	non-portable
Windows uses it a lot	Integrated to the VCL

Now this is a basic list, but, personally, I can only see the point of using a char array when interfacing with windows functions directly. Again, personally, the major benefit I've found from AnsiString is the dynamic allocation.

In Summary

AnsiString is a class, not a data type. Whilst we can treat it in most instances in the same way as a char array, we must be aware that it is a class and not an array. We should, wherever possible use AnsiString when working in BCB, but be aware, it is non-portable, (nor is the VCL), to anything that doesn't define it specifically. Finally, it is easier to use the class member functions than to use the old C style functions, though not quite as quick. (Honestly, you wont notice the speed difference.)

A final point, and a big thank you to Kent here, AnsiString is actually a typedef, defined as follows,
typedef AnsiString String;

This allows us to cheat a little. We can define a variable as a String, (*note case sensitive definition*), and ditch the longwinded AnsiString. That is
String stringone;
is the same as
AnsiString stringone;

Colour Coding:

Information displayed in this colour is information provided by my copy of BCB.
Information displayed in this colour is information you need to type.
Information displayed in this colour is information which is of general interest.

****

Underlying Principals

This tutorial assumes a knowledge of C, and a basic knowledge of BCB. We take a single AnsiString function in turn, in an order which I hope will improve our understanding. I will try to give an example of each, showing how AnsiString does things, and the equivalent C function. Those functions about which I have no clue, are listed at the end, along with those for which I see no point.

How to do AnsiString

Follow the steps below, which are self explanatory.

Step One : Create an application

(a) Create an application form
(b) Add a label, (called label1)
(c) Add another label, (called label2)
(c) Add an edit box, (called Edit1)
(d) Add a button and set properties, name = Action, Caption = &Action.
(e) Hit run and smile :)

The following parts of this tutorial revolve around the AnsiString functions, using the Application for output purposes only.

Functions Covered

This is a description of AnsiString and its functions. Use search, and enter the function you require information about, or work through in order for a more structured understanding. At the end of the document I list the functions of AnsiString I have no clue about.

AnsiString Itself

As I've already mentioned, AnsiString is a class and as such requires a constructor. In this instance, AnsiString contains the following overloaded constructors.

__fastcall AnsiString();
create an empty string. E.g. AnsiString StringOne;
__fastcall AnsiString(const char* src);
create a string from a char array, E.g.

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  String FirstString; // declare an empty AnsiString,  note the
                      // interchange of String for AnsiString

  char *CharArrayOne; // declare a pointer to a char

  CharArrayOne = new char[255];      //allocate memory for the array

  FirstString = "This is test One";  // use overloaded = to provide
                                     // assignment

  strcpy (CharArrayOne,"This is a char array");  // C method of assignment
  String ThirdString("This is another test");
  String FourthString(CharArrayOne);

  delete CharArrayOne;               // free memory pointer
}
```

The above example, whilst useless, shows how a char array and AnsiString differ.
Firstly we declare an empty AnsiString, called FirstString.
Next, we declare a pointer to a char, CharArrayOne. We then have to allocate memory space for the actual array of chars we wish to use.
FirstString = "This is test One" uses the =operator to *assign* our string to our AnsiString.
The next line shows the C assignment method using strcpy to *copy* the string into our CharArrayOne.
ThirdString is assigned its value using the char* constructor, and FourthString uses the same char* constructor to *copy* the value from CharArrayOne.
Finally, we release our memory allocation using delete.

The thing to notice here is how much easier the use of the AnsiString class is to manipulate strings. Even

at this simple level,
AnsiString FirstString; is a lot easier than the C method of

```
char *CharArrayOne;
CharArrayOne = new char[255];
delete CharArrayOne;
```

For the AnsiString we have had to know nothing in advance about the length of the string, memory allocation, both setting and freeing, and with the use of multiple constructors, we don't really have to know too much about what our string will accept as input.

Consider the following code,

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  String ansi_string_array[5];        // declare an array of empty AnsiStrings
  char CharArrayOne[5][255];          // declare an array of char array's of 255
                                      // chars each each does the same,

  ansi_string_array[1] = "This is a string"; // assign the first string as a char*
  ansi_string_array[2] = 25;                 // assign the second string as an int

  //As above in C
  strcpy(CharArrayOne[1],  "This is a string");      // using strcpy to assign a string
  sprintf(CharArrayOne[2],  "%d", 25);               // assign an int and
                                                     // convert to a string.

}
```

Again notice the ease for the AnsiString. If these arrays where assigned to reading in a list of types, in C you would have to provide a handler for each type passed, (i.e. if(int)…, if(char)…., if(char *)…, if(double)….etc.). With AnsiString, you just use the = operators, and allow the constructors and operators to take care of the conversion for you.

The Supported Constructors are;

__fastcall AnsiString();	*//Create an empty string*
__fastcall AnsiString(const char* src);	*//Create a string from a char **
__fastcall AnsiString(const AnsiString& src);	*//Create a string from another AnsiString*

__fastcall AnsiString(const char* src, unsigned char len);

constructs an AnsiString (src) of length len.

e.g.

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  String Test("This is a test",6);    // Construct an AnsiString called test
  Label1->Caption = Test;
}
```

produces a label that says "This I”

__fastcall AnsiString(const wchar_t* src);

wchar_t is a 16 bit char type called WideChar, used for extended character sets.
wchar_t is a ‘C' style string based on the wchar_t.*

(from the Borland Documentation).
“In C++ programs, wchar_t is a fundamental data type that can represent distinct codes for any element of the largest extended character set in any of the supported locales. A wchar_t type is the same size, signedness, and alignment requirement as an int type.”

__fastcall AnsiString(int src); *//Create a string from an int* *e.g.*

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  String Test(6);                    //Create the AnsiString "6"
  Label2->Caption = Test;
}
```

__fastcall AnsiString(double src); *//Create a string from a double.*

The Supported Operators are;

AnsiString& __fastcall operator =(const AnsiString& rhs);
Assigns the value of right hand side to the string on the left hand side. Works on the types for which there is an AnsiString Constructor.
friend AnsiString __fastcall operator +(const char*, const AnsiString& rhs);
Adds, concatenates, two strings together. Again it works on all types for which there is an AnsiString Constructor.
AnsiString& __fastcall operator +=(const AnsiString& rhs);
Assigns and concatenates. Again works on all types for which there is an AnsiString Constructor.

More on AnsiString and char *

Consider the following code

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  for(int x = 1; x < Edit1->Text.Length()+1; x++)
  {
    Label1->Caption = Edit1->Text[x];   // Send the output string one character at a time

    Update();                          // Refresh the display

    Sleep(1000);                       //give us a chance to see it in action
  }
}
```

Now you can see that you've accessed the AnsiString class in the same way as you would an array. It is important to note, you have used the [] as an overloaded operator of the class, and not as an array index, even though the results appear to be what we would expect. *("C" programmers delight or what ?)*
An array of AnsiStrings can be declared as follows,

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{

  AnsiString *MyStrings;              // declare a pointer to the start of the string

  MyStrings = new AnsiString[10];     // allocate memory for it

  for(int x = 1; x < Edit1->Text.Length()+1; x++)
  {
    Label1->Caption = Edit1->Text[x];  // Send the output string one character at a time
                                       // Uses the [] overloaded index operator of
                                        // AnsiString

    MyStrings[x] = Edit1->Text[x];  // equate the our string array to the caption

       //MyStrings uses a standard index array.
       //Edit1->Text[x] uses the AnsiString overloaded function

    Form1->Caption = MyStrings[x];  //show our string array in the forms caption

    Update();  // update the forms display.  Required because we're inside the button
```

```
               // function

    Sleep(1000);        //give us a chance to see it in action
  }
}
```

As you can now see, we have an array of AnsiStrings, in this case a maximum of 11 strings, which will end up being 1 character in length, but could actually be any length. *(If you try to go beyond the 11 strings declared, you'll create an access violation.)*

Take care with the example above, and notice the difference between an array of AnsiString(s) and the AnsiString[] overloaded operator. Run the app above, and hit the action button. You should see the caption of label1 and the forms caption display Edit1 a character at a time. Now enter the following string into the edit box, "*ThisIsAStri#ngLongerThanElevenCharacters*". Hit the action button again, and the displays will start through the string a character at a time again. However, when it reaches the #, the application will throw an exception, you have overstepped the bounds of your array of AnsiStrings; Crunch!

(*accept the error and quit the app normally. If you have the IDE handling exceptions, press ctrl F2 to reset the application.)*

Make sure you grasp this difference. It is very important to note the difference between an array of AnsiStrings and the index of an AnsiString. Pay attention to the next section which will show the other major difference of an AnsiString and an indexed array of Chars.

Now notice the function called as part of the for loop definition, Edit1->Text.Length(). As Edit1->Text is an AnsiString, it has access to all the functions that an AnsiString does, this one returning an integer of the length of the string. Here is how the "C" function strlen equates to AnsiString.Length(),

Length()
"C" equivalent, strlen().

To prove this, enter the following code,

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  char *str;                  //Yes a "C" style string
  int AnsiLength,  CLength;

  str = new char[255]; //allocate memory for our C char array

  AnsiLength = Edit1->Text.Length();         // Put the length of the text
                                             // into AnsiLength

  strcpy(str, Edit1->Text.c_str());          // Use the AnsiString c_str()
                                             // function to return a char *

  CLength = strlen(str);                     //Get the length of the C string

  Form1->Caption = AnsiLength;               //Display the AnsiLength value,
  Label1->Caption = CLength;                 //Display the C string length

  delete str;                                //free memory allocation
}
```

Notice how both the lengths are the same, though what did you really expect. The function c_str() returns the "C" equivalent of AnsiString, ***but they are not the same***, add the following code at the end of the above to prove this, (**note, move the delete str; line to the end**)

```
Update();                             //update the screen
```

```
Sleep(2000);                         //rest for two seconds
for(int x = 1; x < AnsiLength + 1; x++)
{
  Form1->Caption = Edit1->Text[x];           // Display AnsiString a
                                             // character at a time

  Label1->Caption = str[x-1]; // Display C String a character at a time

  Update();                 // force screen update
  Sleep(1000);              // give us chance for a look
}
```

The thing to note here is the difference in the index of the array. An AnsiString starts with its base as 1 upwards, where as a "C" string has its base as 0. If you're converting from "C" to AnsiString, it is very easy to get bounds errors by starting a string index at 0 when you're using an AnsiString, you have been warned.

There's More !

AnsiString provides most of the "C" equivalent functions as members of its class. The list below gives the AnsiString function and the "C" equivalent. Play with them in the above manner, and you'll find that AnsiString really is a class we should have no difficulty in working with. Also take note of the functions for which "C" has no simple equivalent.

Functions.

First the easy ones:

enum TStringFloatFormat {sffGeneral, sffExponent, sffFixed, sffNumber, sffCurrency };
The above enum is used in the following function
static AnsiString __fastcall FloatToStrF(long double value, TStringFloatFormat format, int precision, int digits);
static AnsiString __fastcall Format(const AnsiString& format,const TVarRec *args, int size);
static AnsiString __fastcall FormatFloat(const AnsiString& format,const long double& value);
static AnsiString __fastcall IntToHex(int value, int digits);
static AnsiString __fastcall CurrToStr(Currency value);
static AnsiString __fastcall CurrToStrF(Currency value, TStringFloatFormat format, int digits);

The above functions all appear to work in a similar manner. By way of example, FloatToStrF, This takes the number displayed and converts it into the Windows formats that equate through the Localisation configuration of the users machine,

E.g.

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  long double Number, Number1;

  Number = 21.27374757;
  AnsiString String;

  Form1->Caption = String.FloatToStrF(Number,AnsiString::sffGeneral,3,3);

  Sleep(1000);

  Form1->Caption = String.FloatToStrF(Number,AnsiString::sffExponent,4, 4);

  Sleep(1000);

  Form1->Caption = String.FloatToStrF(Number,AnsiString::sffFixed,3,3);
  Sleep(1000);

  Form1->Caption = String.FloatToStrF(Number,AnsiString::sffNumber,5,4);

  Sleep(1000);

  Form1->Caption = String.FloatToStrF(Number,AnsiString::sffCurrency,4, 2);

  Sleep(1000);
}
```

Notice the different formats of the same number, especially the sffCurrency one.

friend AnsiString __fastcall operator +(const char*, const AnsiString& rhs);

This is a simple addition of two strings, concatenation.
E.g.

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  AnsiString SillyString;

  SillyString = Label1->Caption + Edit1->Text;
  Label1->Caption = Edit1->Text;
  Form1->Caption = SillyString;
}
```

The use of SillyString isn't really necessary, Form1->Caption = Label1->Caption + Edit1->Text would work just as well.

Note: Press the button twice and the string builds up from, *Label1Edit1*, to, *Edit1Edit1*

Now Change SillyString to, SillyString = Form1->Caption + Label1->Caption + Edit1->Text; and press the button a few times. You have just performed a multiple concatenation.

static AnsiString __fastcall StringOfChar(char ch, int count);

Convert a string to contain a string of *count characters ch*. Pay attention to the way it is done, it has to be assigned from one string to another.
e.g.

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  AnsiString SillyString;
  Form1->Caption = SillyString.StringOfChar('x', 10);
}
```

void __fastcall Insert(const AnsiString& str, int index);

Insert a string into a string at a particular point. In the following example, notice again how the function works, *Form1->Caption.Insert(str,int) doesn't work, and Form1->Caption = Form1->Caption.Insert… throws a not an allowed type error.*

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  AnsiString Test;

  Test = Form1->Caption;
  Test.Insert(Label1->Caption,  2);
  Form1->Caption = Test;
}
```

void __fastcall Delete(int index, int count);

Delete count characters from the string, starting at index. If the index is a position outside the range of the string, no action is taken. (press the button a few times to prove it.)

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  AnsiString Test;

  Test = Form1->Caption;
  Test.Delete(2,1);
  Form1->Caption = Test;
}
```

AnsiString& __fastcall operator =(const AnsiString& rhs);

Simple equate. We've used it already quite a lot. The only thing to notice is the const for the right hand side of the equation, as in Form1->Caption = Test, ***not*** Form1->Caption = Test.Delete(2,1);

AnsiString& __fastcall operator +=(const AnsiString& rhs);
The operator +=() is not supported on a property with getter/setter functions.
(Whatever that means?)
As above, note the constant but you can say, Form1->Caption += Test; to replace Form1->Caption = Form1->Caption + Test;

<u>Comparisons</u>

bool __fastcall operator ==(const AnsiString& rhs) const;
bool __fastcall operator !=(const AnsiString& rhs) const;
bool __fastcall operator <(const AnsiString& rhs) const;
bool __fastcall operator >(const AnsiString& rhs) const;
bool __fastcall operator <=(const AnsiString& rhs) const;
bool __fastcall operator >=(const AnsiString& rhs) const;
int __fastcall AnsiCompare(const AnsiString& rhs) const;
int __fastcall AnsiCompareIC(const AnsiString& rhs) const; //ignorecase

All fairly straightforward, just pay attention to the constant use again.
i.e.

```
{
  AnsiString stOne, stTwo, stThree;

  stOne = "One";
  stTwo = "Two";
  stThree = "Three";

  if(stOne == stTwo)   //this is okay,

  if(stOne == (stTwo.Insert(stThree, 2));    //Not Okay.
}
```

bool __fastcall IsEmpty() const;

Equivalent of the old "C" empty sting,

"C"	AnsiString
if(string == "\0")	If(string.IsEmpty())

int __fastcall Pos(const AnsiString& subStr) const;

Returns the position of the first occurrence of the substring within the AnsiString.

AnsiString __fastcall LowerCase() const;
AnsiString __fastcall UpperCase() const;
Returns the string converted to upper or lower case characters. TestString = TestStrong.LowerCase();

AnsiString __fastcall Trim() const;
Use the Trim function to remove blank space before and after the first printing character.
AnsiString __fastcall TrimLeft() const;
Use the Trim function to remove blank space before the first printing character.
AnsiString __fastcall TrimRight() const;
Use the Trim function to remove blank space after the first printing character.

AnsiString __fastcall SubString(int index, int count) const;
Returns a sub-string from the AnsiString, starting at position *index*, and of *count* characters in length.
e.g. If Edit1->Text = "Edit1" the following would place "dit" into Label1->Caption

Label1->Caption = Edit1->Text.SubString(2,3);

int __fastcall ToInt() const;
Convert to an integer, equivalent of "C" function atoi()

int *variable* = AnsiString *String*.ToInt();

double __fastcall ToDouble() const;
Convert to a double, as above

void __fastcall SetLength(int newLength);
SetLength sets the length of a string.
Eg.

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  String Test("This is a test");
  Test.SetLength(6);
  Label1->Caption = Test;
}
```

produces a label that says "This i".

void __fastcall Unique();
Use this function to make a string unique (refcnt == 1).

Although I am not positive about this, if AnsiString works the same way as cstring does it stores strings which are the same in the same location. And only creates a new location if they are changed. I believe unique ensures that even if the strings are the same they are stored in different locations.
e.g.

```
String Test1 = "Hello World!";
String Test2 = "Hello World!";    // Both Test1 & Test2 would point
                                  // to the same memory address.
Test1 = "New String";             // Now they would point to different Addresses.
```

int __fastcall WideCharBufSize() const;

Determines the number of WideChars needed to hold the current AnsiString.

wchar_t* __fastcall WideChar(wchar_t* dest, int destSize) const;

Returns the AnsiString in WideChar format...
e.g.

```
void __fastcall TForm1::ActionClick(TObject *Sender)
{
  String AString = "HelloWorld!";
  int Length = AString.WideCharBufSize();
  wchar_t* WCString = new wchar_t[Length];
  AString.WideChar(WCString,Length);
  String Test3(WCString);
  Label1->Caption = Test3;
}
```

****

The following functions will be defined in a later version of this document...

Functions
static AnsiString __fastcall LoadStr(int ident);
static AnsiString __fastcall FmtLoadStr(int ident, const TVarRec *args, int size);
int __fastcall ToIntDef(int defaultValue) const;

MultiByte support
enum TStringMbcsByteType { mbSingleByte, mbLeadByte, mbTrailByte };
TStringMbcsByteType __fastcall ByteType(int index) const;
bool __fastcall IsDelimiter(const AnsiString& delimiters, int index) const;
bool __fastcall IsPathDelimiter(int index) const;
int __fastcall LastDelimiter(const AnsiString& delimiters) const;
bool __fastcall IsLeadByte(int index) const;
bool __fastcall IsTrailByte(int index) const;
int __fastcall AnsiPos(const AnsiString& subStr) const;
char* __fastcall AnsiLastChar() const;

A Final Note

No liability is accepted by the author(s) for anything which may occur whilst following this tutorial