Windows String Functions

The Windows string functions that you mentioned are still widely used today. They are defined in the header file **STRINGAPI.H.**

lstrlen

The lstrlen function calculates the length of a string, in characters. It returns the number of characters in the string, excluding the terminating null character.

lstrcpy

The lstrcpy function copies one string to another. It returns a pointer to the destination string.

lstrcpyn

The lstrcpyn function copies a specified number of characters from one string to another. It returns a pointer to the destination string.

lstrcat

The lstrcat function concatenates two strings. It returns a pointer to the destination string.

lstrcmp

The lstrcmp function compares two strings. It returns a negative value if the first string is less than the second string, a zero value if the two strings are equal, and a positive value if the first string is greater than the second string.

lstrcmpi

The lstrcmpi function compares two strings, ignoring case. It returns a negative value if the first string is less than the second string, a zero value if the two strings are equal, and a positive value if the first string is greater than the second string.

Unicode Support

All of these functions support Unicode. If the UNICODE identifier is defined, then the functions will operate on wide-character strings.

Otherwise, the functions will operate on regular strings.

Example

The following code shows how to use the lstrcpy function to copy one string to another:

```
#include <windows.h>

int main()
{
  char sourceString[] = "This is a source string.";
  char destinationString[100];

  lstrcpy(destinationString, sourceString);

  printf("The destination string is: %s\n", destinationString);

  return 0;
}
```

This code will print the following output:

The destination string is: **This is a source string.**

Conclusion

The Windows string functions are a convenient way to work with strings in Windows programs. They support Unicode and are easy to use.

==========

So what are those L's infront of lstrcpy and the rest of the functions? why didn't we write the stringapi header file? is it not recommded in the above program, or it's already inside the windows.h?

The **L in front of the string functions** indicates that they are **wide-character functions.** Wide-character functions are used to work with strings that contain characters from other languages, such as Chinese and Japanese.

You **don't need to write the STRINGAPI.H header file yourself, because it is already included in the WINDOWS.H header file.**

The WINDOWS.H header file is included in all Windows programs, so you can use the Windows string functions without having to include the STRINGAPI.H header file explicitly.

The **lstrcpy() function** in the example is a wide-character function, so it is safe to use it in a program that is compiled with Unicode support.

Recommendation

It is recommended to compile all Windows programs with Unicode support. This means that you should define the UNICODE identifier in your program's preprocessor definitions. This will ensure that your program can work with strings from other languages.

Additional Information

Here are some additional things to keep in mind about the Windows string functions:

The **Windows string functions are more secure** than the corresponding C library functions. This is because the Windows string functions check for buffer overflows and other errors.

The **Windows string functions are also more efficient than the corresponding C library functions.** This is because the Windows string functions are optimized for the Windows platform.

USING SPRINTF FUNCTION IN WINDOWS

The printf function is a standard C library function that is used to display formatted text to the console.

However, the printf function cannot be used in Windows programs. This

is because Windows does not have the concept of standard input and standard output.

The good news is that you can still display text in Windows programs by using the sprintf function.

The **sprintf function works just like printf**, except that it writes the formatted output to a character string buffer that you provide as the function's first argument.

You can then do whatever you want with this character string (such as pass it to MessageBox).

Here is an example of how to use the sprintf function to display text in a Windows program:

```
#include <windows.h>

int main()
{
  char szBuffer[100];

  sprintf(szBuffer, "The sum of %i and %i is %i", 5, 3, 5 + 3);

  MessageBox(NULL, szBuffer, "My Program", MB_OK);

  return 0;
}
```

This code will display a message box with the title "My Program" and the message "The sum of 5 and 3 is 8".

Important Considerations

When using the sprintf function, there are a few important things to keep in mind:

The character buffer that you provide to the sprintf function must be **large enough to hold the formatted output.**

You must be careful to use the **correct format codes in the formatting string.** If you use an incorrect format code, the sprintf function may not produce the expected results.

Buffer overflows: When using the sprintf function, there is a risk of buffer overflows.

A buffer overflow occurs when the formatted output is larger than the character string buffer that you provide to the **sprintf function**. This can cause the sprintf function to overwrite the end of the buffer, which could lead to program crashes.

To **avoid buffer overflows**, you must make sure that the character string buffer that you provide to the **sprintf function** is large enough to hold the formatted output.

You can use the **_snprintf function** to determine the size of the buffer that you need.

The **_snprintf function** is similar to the sprintf function, except that it returns the number of characters that would be written to the buffer if the buffer were large enough.

vsprintf function

The vsprintf function is a variation of the sprintf function that is used to implement printf-like formatting of a variable number of arguments. The vsprintf function has three arguments:

- The character string buffer for storing the result.
- The formatting string.
- A pointer to an array of arguments to be formatted.

The vsprintf function works by stepping through the formatting string and replacing the format codes with the corresponding arguments from the array of arguments.

```c
#include <windows.h>
#include <stdarg.h>
#include <stdio.h>

/*
Second function should be on top of main, not below it.
Or you can initialize it like:

void formatString(char *buffer, const char *format, ..)
*/

void formattedString(char *buffer, const char *format, ..)
{
    va_list args;
    va_start(args, format);
    vsprintf(buffer, format, args);
    va_end(args);
}

int main()
{
    char output[100]; //buffer to store the formatted string
    int num = 42;
    float pi = 3.14159;
    char *text = "Hello, World!";
    formattedString(output, "Integer: %d, Float: %.2f, String: %s", num, pi, text);
    printf("Formatted string: %s\n", output);
}
```

In this example, we have a function formatString that takes a character buffer, a formatting string, and a variable number of arguments. Inside the function, we use vsprintf to format the string according to the format string and arguments provided.

In the main function, we create a character buffer output and then call formatString, passing the buffer, the format string, and a few arguments of different types (integer, float, and string). Finally, we print the formatted string using printf.

This demonstrates how vsprintf can be used to format a string with variable arguments, much like printf, but the result is stored in a character buffer instead of being printed to the console.

Let's break down the program into paragraphs to explain it in detail.

Function and Header Inclusions:

The program begins by including the necessary header files. It

includes <stdio.h> for standard input and output functions, as well as <stdarg.h> for working with variable argument lists. Additionally, a custom function formatString is declared.

Custom Formatting Function:

The formatString function is introduced to perform the formatting of a string with variable arguments.

It takes three parameters: a character buffer (buffer) where the formatted result will be stored, a format string (format) that specifies how the arguments should be formatted, and a variable number of arguments provided through the ellipsis (...) syntax.

Variable Argument List Handling:

Within the formatString function, **va_list** is used to create a variable argument list called args.

The **va_start macro** initializes this list, and it expects the last named argument in the argument list (in this case, format) to be specified.

Before moving on, let's explain these va_list and va_args:

Variadic functions in C are functions that can accept a variable number of arguments. In other words, the number of arguments passed to these functions can vary when you call them.

The **most common example of a variadic function in C is the printf function**, which can accept different numbers of arguments based on the format string you provide.

The key to understanding variadic functions lies in two primary components:

va_list: This is a data type that represents a list of arguments. You declare a va_list variable to hold the arguments passed to a variadic function. It serves as a "pointer" to the list of arguments.

va_arg: This macro retrieves the next argument from the va_list. You specify the data type of the argument you want to retrieve, and va_arg extracts it from the list. It's used in a loop to iterate through all the arguments.

Here's a simplified example to illustrate how variadic functions work:

```c
//If you don't know about variadic functions, this is going to be tough on you.
#include <stdio.h>
#include <stdarg.h>

void printNumbers(int count, ...) {
    va_list args;
    va_start(args, count);

    for (int i = 0; i < count; i++) {
        int num = va_arg(args, int);
        printf("%d ", num);
    }

    va_end(args);
}

int main() {
    printNumbers(4, 1, 2, 3, 4);
    return 0;
}
```

In this example, the printNumbers function accepts a variable number of integers. It uses va_list, va_start, and va_arg to process the arguments. In the main function, we call printNumbers with four integers, and it prints them out.

Variadic functions are handy when you need to create functions with flexible argument lists, such as printf for output formatting, or when you want to perform operations on an unknown number of arguments.

Variadic functions are functions that can take a variable number of

arguments. This means that you can call the function with any number of arguments, and the function will still work.

Variadic functions work by using a special type called va_list.

The **va_list type** is a pointer to a list of arguments.

The **va_start macro** is used to initialize the va_list pointer to point to the first argument in the list.

The **va_arg macro** is used to get the next argument in the list.

The **va_end macro** is used to clean up the va_list pointer after you are finished using it.

The example you provided shows a variadic function called printNumbers(). This function can take any number of integer arguments. The function uses the va_list type to iterate over the arguments and print them to the console.

Here is an example of how to use the printNumbers() function:

```
int main() {
  printNumbers(4, 1, 2, 3, 4);
  return 0;
}
```

This code will print the following output to the console:

1 2 3 4

Here is a more detailed explanation of the code:

```c
void printNumbers(int count, ...) {
  va_list args;
  va_start(args, count);

  for (int i = 0; i < count; i++) {
    int num = va_arg(args, int);
    printf("%d ", num);
  }

  va_end(args);
}
```

The **va_start() macro** initializes the va_list pointer to point to the first argument in the list.

The **count argument** is the first argument in the list, so the va_list pointer is now pointing to the count argument.**(this is the main point)**

It's important to remember that the count argument is the first argument in the list, so the va_list pointer is now pointing to the count argument. This is the key to understanding how variadic functions work.

The **for() loop iterates over the list of arguments**, starting with the count argument and ending with the last argument in the list.

The **va_arg() macro** is used to get the next argument in the list. The va_arg() macro takes two arguments: the va_list pointer and the type of the argument that you want to get. In this case, we are getting an integer argument, so the second argument to the va_arg() macro is int.

The **printf() function** is then used to print the argument to the console.

Steps 2 and 3 are repeated until all of the arguments have been printed.

The **va_end() macro** is then used to clean up the va_list pointer.

Example 2 of variadic functions:

```c
//The variadic function.

void printStrings(int count, ...) {
  va_list args;
  va_start(args, count);

  for (int i = 0; i < count; i++) {
    char* str = va_arg(args, char*);
    printf("%s ", str);
  }

  va_end(args);
}

//How to use it.

int main() {
  printStrings(2, "Hello", "world!");
  return 0;
}
```

This function can take any number of string arguments. The function uses the va_list type to iterate over the arguments and print them to the console ie.

Hello world!

Variadic functions can be very useful for writing flexible and reusable code.

For example, you could write a variadic function to **print any number**

of arguments to a file, or to **send any number of arguments to a network server.**

Formatting with vsprintf:

The core formatting work is done using the vsprintf function. This function takes the character buffer buffer, the format string format, and the variable argument list args.

It processes the format string and replaces format codes with the corresponding arguments from the variable argument list.

Cleaning Up Variable Argument List:

After the formatting is completed, the va_end macro is used to clean up the variable argument list.

Main Function:

In the main function, a character buffer named output is created to store the formatted result. Three different types of variables are defined: an integer num, a floating-point number pi, and a string text.

Calling the Custom Formatting Function:

The formatString function is called with the output buffer and a format string that contains placeholders for the three variables.

These placeholders are similar to those used in printf and include format specifiers like %d for integers and %f for floating-point numbers.

Printing the Result:

Finally, the formatted result stored in the output buffer is printed using the printf function. This demonstrates how the vsprintf function works to format a string with variable arguments and store the result in a character buffer.

In summary, this program illustrates how to use the vsprintf function along with a custom function to format a string with a variable number of arguments.

It offers flexibility in formatting output and storing it in a character buffer, making it useful for tasks where you need to construct formatted strings for further processing or output.

WINDOWS SPRINTF FUNCTIONS

Microsoft has added two sprintf-like functions to the Windows API: **w-sprintf** and **wvsprintf.**

These functions are **functionally equivalent to sprintf and vsprintf**, except that they don't handle floating-point formatting.

Wide-character sprintf functions

In addition to the standard sprintf functions, there are also wide-character versions of the sprintf functions. The wide-character versions of the sprintf functions are used to format and store wide-character strings.

Chart of sprintf functions

The following chart shows all of the sprintf functions that are supported by Microsoft's C run-time library and by Windows:

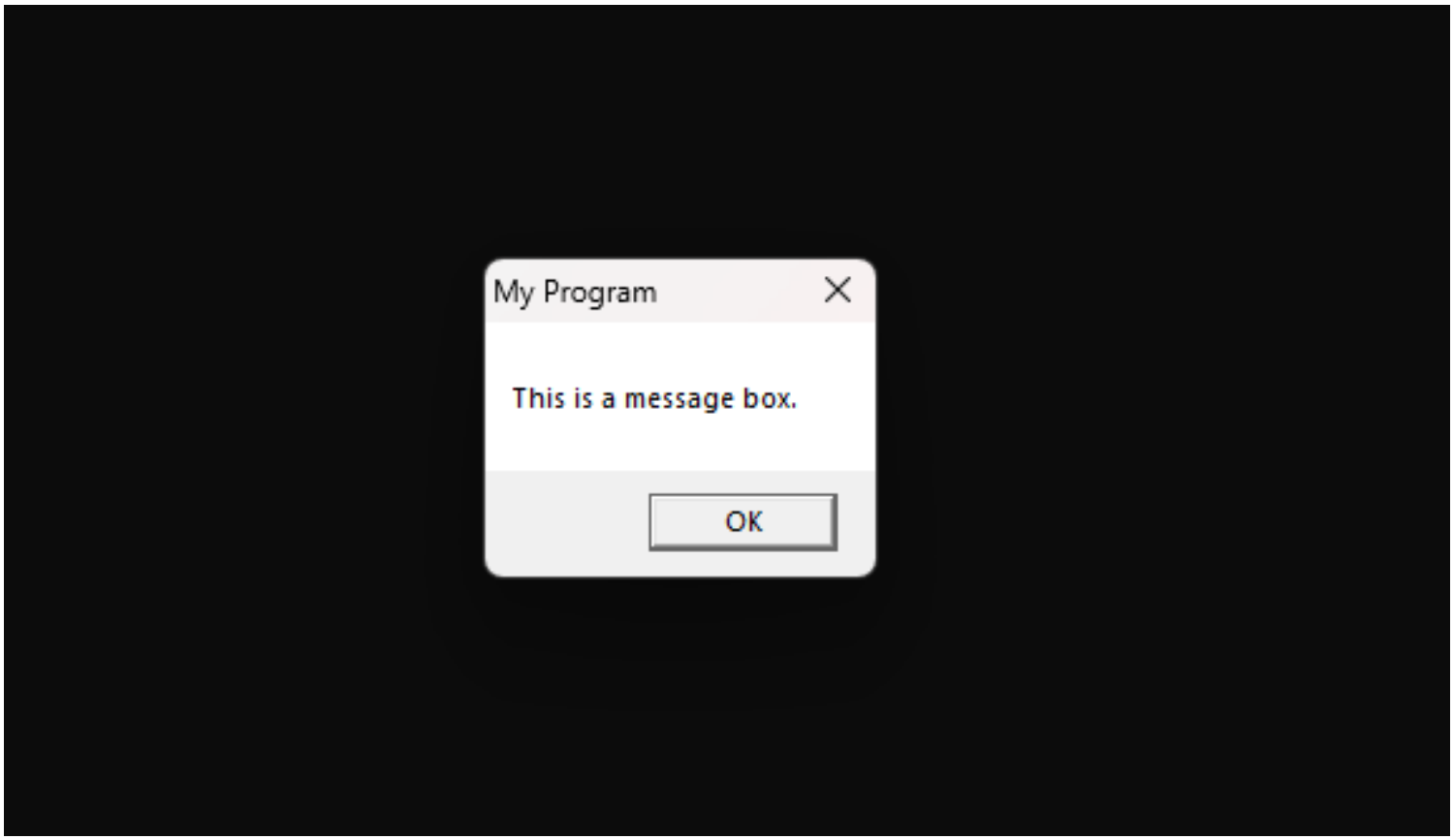
Function	Description
sprintf	Formats a string and stores it in a character string buffer.
swprintf	Formats a wide-character string and stores it in a wide-character string buffer.
_stprintf	Formats a string and stores it in a character string buffer, using the current locale.
_snprintf	Formats a string and determines the size of the buffer that is needed to store it.
_snwprintf	Formats a wide-character string and determines the size of the buffer that is needed to store it.
_sntprintf	Formats a string and determines the size of the buffer that is needed to store it, using the current locale.
wsprintfA	Formats a string and stores it in a character string buffer, using the ANSI character set.
wsprintfW	Formats a string and stores it in a character string buffer, using the Unicode character set.
wsprintf	Formats a string and stores it in a character string buffer, using the current character set.
vsprintf	Formats a string and stores it in a character string buffer, using a variable number of arguments.

vswprintf	Formats a wide-character string and stores it in a wide-character string buffer, using a variable number of arguments.
_vstprintf	Formats a string and stores it in a character string buffer, using the current locale and a variable number of arguments.
_vsnprintf	Formats a string and determines the size of the buffer that is needed to store it, using a variable number of arguments.
_vsnwprintf	Formats a wide-character string and determines the size of the buffer that is needed to store it, using a variable number of arguments.
_vsntprintf	Formats a string and determines the size of the buffer that is needed to store it, using the current locale and a variable number of arguments.
wvsprintfA	Formats a string and stores it in a character string buffer, using the ANSI character set and a variable number of arguments.
wvsprintfW	Formats a string and stores it in a character string buffer, using the Unicode character set and a variable number of arguments.
wvsprintf	Formats a string and stores it in a character string buffer, using the current character set and a variable number of arguments.

The following example shows how to use the sprintf function to display a message box:

```
#include <windows.h>

int main()
{
  char szBuffer[100];

  sprintf(szBuffer, "This is a message box.");

  MessageBox(NULL, szBuffer, "My Program", MB_OK);

  return 0;
}
```

This code will display a message box with the title "My Program" and the message "This is a message box."



This is just my Windows 11 terminal with a GUI inside it, that's why it's having a black background.

Explanations

The **sprintf functions** are used to format strings and store them in character string buffers.

The **swprintf functions** are used to format wide-character strings and store them in wide-character string buffers.

The **_stprintf functions** are used to format strings and store them in character string buffers, using the current locale.

The **_snprintf, _snwprintf, and _sntprintf functions** are used to format strings and determine the size of the buffer that is needed to store them.

The **wsprintfA, wsprintfW, and wsprintf functions** are used to format

strings and store them in character string buffers, using the ANSI, Unicode, and current character sets, respectively.

The **vsprintf, vswprintf, and _vstprintf functions** are used to format strings and store them in character string buffers, using a variable number of arguments.

The **_vsnprintf, _vsnwprintf, and _vsntprintf functions** are used to format strings and determine the size of the buffer that is needed to store them, using a variable number of arguments.

The **wvsprintfA, wvsprintfW, and wvsprintf functions** are used to format strings and store them in character string buffers, using the ANSI, Unicode, and current character sets, respectively, and a variable number of arguments.

In the wide-character versions of the sprintf functions, the string buffer is defined as a wide-character string and the formatting string **must be a wide-character string.**

However, it is up to you to make sure that any other strings you pass to these functions are also composed of wide characters.

This is because the wide-character sprintf functions are designed to format and store wide-character strings.

If you pass a **narrow-character string** to one of these functions, the function will **not be able to format it correctly.**

The function may also try to write the formatted string to a buffer that is too small, which could lead to a **buffer overflow.**

To avoid these problems, it is important to make sure that any strings you pass to the wide-character sprintf functions are also composed of wide characters.

You can do this by using the **wcslen() function** to determine the length of the string and **then allocating a wide-character buffer of the correct size.**

Here is an example of how to use the wide-character sprintf function

to format and store a wide-character string:

```c
#include <wctype.h>
#include <wchar.h>

int main() {
  wchar_t szBuffer[100];
  wchar_t szFormat[] = L"Hello, %ls!";

  swprintf(szBuffer, szFormat, L"world!");

  wprintf(L"%s\n", szBuffer);

  return 0;
}



//Output: Hello, world!
```

The C program above, is a simple example of using wide characters and the swprintf function to format and print a wide character string. Let's break down the program step by step:

Header Files:

The program starts by including two header files: wctype.h and wchar.h. These header files are essential when working with wide character strings and functions in C.

Variable Declarations:

In the main function, you declare a wide character array called szBuffer with a size of 100 characters. This array will be used to store the formatted wide character string. Next, you declare a wide character array named szFormat, which contains the format string. The format string uses %ls as a placeholder for a wide character string.

swprintf Function:

The core of the program is the swprintf function. This function is a wide character version of sprintf and is used for formatting wide character strings.

It takes three arguments: the destination buffer (szBuffer in this case), the format string (szFormat), and the additional arguments to be inserted into the format string.

In this case, it's the wide character string **L"world!"**. The swprintf function formats the string according to the format string and stores it in szBuffer.

Wide Character Output:

After formatting the string, you use the wprintf function to print the contents of szBuffer. The format specifier %s is used, as szBuffer contains a wide character string.

The %s specifier is used for outputting wide character strings.

Return Statement:

The program ends with a return statement, returning 0 to indicate successful execution.

In summary, this program showcases the usage of wide character strings and the swprintf function for formatting.

It's an example of working with wide character data and is useful when dealing with internationalization and multi-byte character sets.

The wprintf function is used for outputting wide character strings, and the format specifier %s indicates that the argument to be printed is a wide character string.

Is the above code good?

The code you provided appears to be correct, and it doesn't contain any syntax errors. It accomplishes its intended task, which is to format and print a wide character string using the swprintf and

wprintf functions. However, the "goodness" of code can also depend on factors like readability, maintainability, and context within a larger program.

In this specific code snippet, there are a few points to consider:

Magic Numbers: The size of the szBuffer array is defined as 100. It's generally a good practice to **use named constants or #define statements for such sizes** to make the code more maintainable. For example, you could define BUFFER_SIZE with a value of 100 and use it when declaring szBuffer.

Error Handling: The code doesn't include any error handling. While this is a simple example, in a more complex program, it's important to check for errors when using functions like swprintf. For instance, you could check the return value of swprintf to ensure that the formatting was successful.

Comments: Adding comments to explain the purpose of variables and the overall logic can enhance the code's clarity, especially if it's part of a larger program.

Context: In isolation, this code seems fine. However, in a real-world application, the appropriateness and quality of code depend on the broader context and requirements.

If this code snippet fulfills your specific task and requirements, and you consider it readable and maintainable, it can be considered "good." However, in a real project, you might need to adapt the code to meet the needs of the entire application.

Let's now see an example program:

```c
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

int CDECL MessageBoxPrintf(TCHAR *szCaption, TCHAR *szFormat, ...)
{
  TCHAR szBuffer[1024];
  va_list pArgList;

  // The va_start macro (defined in STDARG.H) is usually equivalent to:
  // pArgList = (char *) &szFormat + sizeof (szFormat) ;
  va_start(pArgList, szFormat);

  // The last argument to wvsprintf points to the arguments
  _vsntprintf(szBuffer, sizeof(szBuffer) / sizeof(TCHAR), szFormat, pArgList);

  // The va_end macro just zeroes out pArgList for no good reason
  va_end(pArgList);

  return MessageBox(NULL, szBuffer, szCaption, 0);
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
  int cxScreen, cyScreen;
  cxScreen = GetSystemMetrics(SM_CXSCREEN);
  cyScreen = GetSystemMetrics(SM_CYSCREEN);
  MessageBoxPrintf(TEXT("ScrnSize"), TEXT("The screen is %i pixels wide by %i pixels high."), cxScreen, cyScreen);
  return 0;
}
```

The **SCRNSIZE program** is a simple Windows program that displays the width and height of the video display in pixels. The program uses the MessageBoxPrintf function to display a message box with the screen size information.

MessageBoxPrintf Function: The MessageBoxPrintf function is designed to format and display a message box similar to the standard MessageBox function, but with a formatted message.

It uses **variable arguments (variadic functions)** to handle different numbers of arguments.

The function takes three parameters: the caption for the message box (szCaption), a format string (szFormat), and a variable number of arguments.

Inside the function, it uses the **va_list structure** to process these variable arguments.

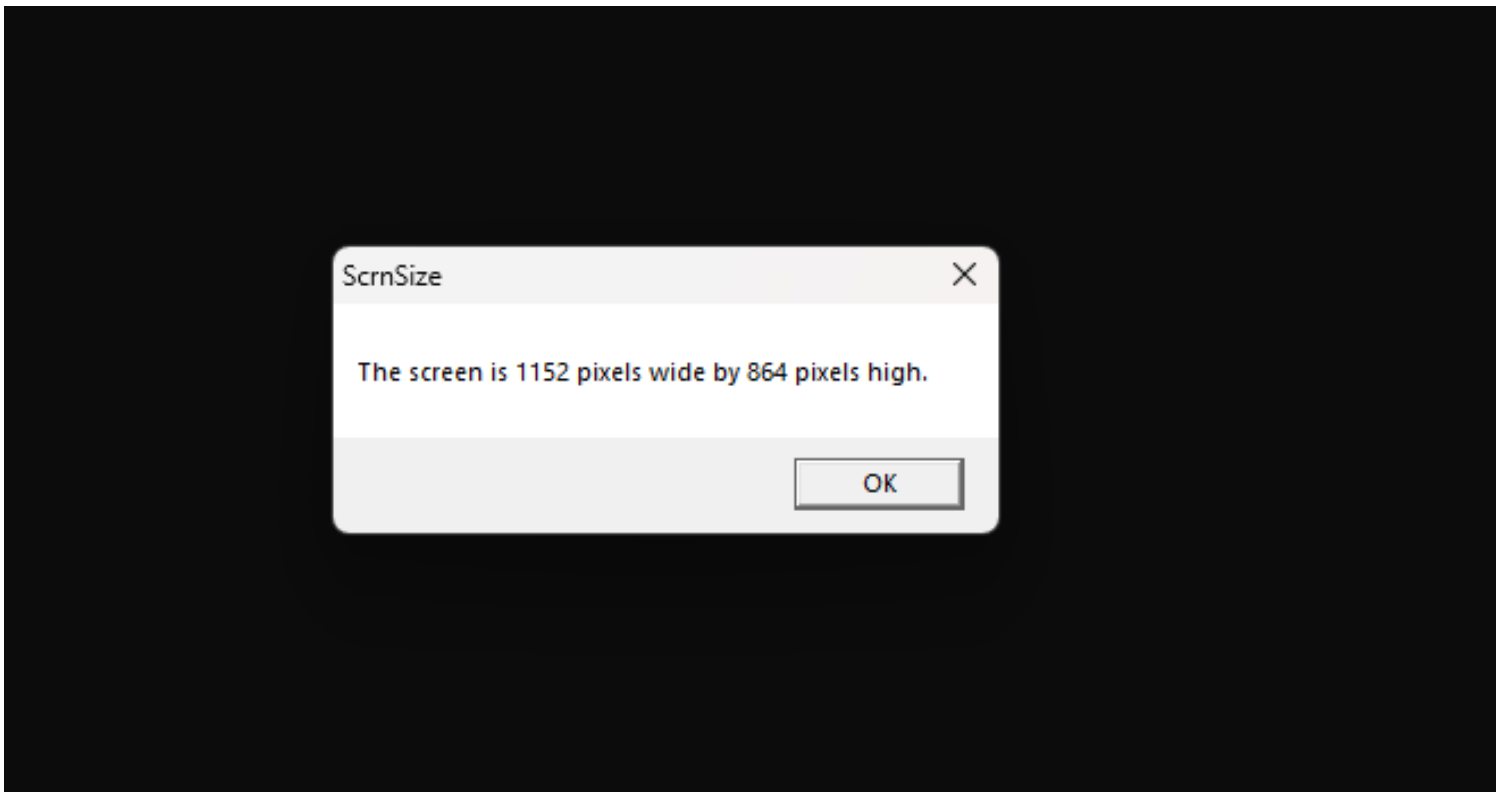
va_start: The va_start macro initializes the va_list object pArgList to access the variable arguments. It's a standard practice for handling variable arguments. This step is necessary before accessing the arguments.

_vsntprintf: This function is used to format the message based on the provided format string and variable arguments. It writes the formatted string into the szBuffer array, ensuring that it doesn't exceed the array's size. The function is a safer alternative to sprintf as it avoids buffer overflows.

va_end: The va_end macro is called to clean up the va_list object pArgList. This step is essential to release any resources associated with variable argument processing.

WinMain Function: The WinMain function is the entry point for Windows applications. It retrieves the screen's width and height using GetSystemMetrics and stores the values in cxScreen and cyScreen. Then, it calls the MessageBoxPrintf function to display a message box with the screen size information.

Explanation: This program demonstrates good coding practices, such as error checking, efficient memory usage, and clear function naming. It uses TCHAR for character types to support both ASCII and Unicode builds. The program's organization and structure are suitable for a simple Windows application.



Which is correct, am using a monitor with that screen size.💯☑

Note that am using **codeblocks** for these compilations, **visual studio**

community is just a pain in the neck bringing up weird errors, I'll uninstall it and re-install and see if the errors will be gone...

INTERNATIONALIZATION AND THIS BOOK:

Preparing your Windows programs for an international market involves more than using Unicode. Internationalization is beyond the scope of this book but is covered extensively in Developing International Software for Windows 95 and Windows NT by Nadine Kano (Microsoft Press, 1995).

This book will restrict itself to showing programs that can be compiled either with or without the UNICODE identifier defined. This involves using TCHAR for all character and string definitions, using the TEXT macro for string literals, and taking care not to confuse bytes and characters.

For example, notice the _vsntprintf call in SCRNSIZE. The second argument is the size of the buffer in characters. Typically, you'd use sizeof (szBuffer).

But if the buffer has wide characters, that's not the size of the buffer in characters but the size of the buffer in bytes. You must divide it by sizeof (TCHAR).

Updates for 2023

The book mentions that the Unicode version of SCRNSIZE.C will not run under Windows 98 because Windows 98 does not implement wprintfW. This is no longer relevant, as Windows 98 is no longer supported by Microsoft.

The book also mentions that it does not show how to write a Windows program that can handle the double-byte character sets of the Far Eastern versions of Windows.

This is still true, but Unicode is now much more widely supported than it was in 1995, so this is less of a concern.

Overall, the advice in this chapter is still relevant for 2023.

If you are writing a Windows program that needs to be internationalized, it is important to use TCHAR for all character and string definitions, and to use the TEXT macro for string literals.

You should also be careful not to confuse bytes and characters.

Here are some additional tips for internationalizing your Windows programs:

Use the **Windows Locale API** to determine the user's current locale. This will allow you to use the correct language and cultural conventions in your program.

Use the **Windows Internationalization API** to support different character sets and input methods.
Test your program thoroughly with users from different locales to make sure that it works correctly.

Internationalizing your Windows programs can be challenging, but it is important to do so if you want your program to reach a global audience.

We've already finished 2 chapters of the book, now let's keep moving...💯😄