Simple Dynamic Allocation and Pointers in C

The C Standard Library provides a small collection of functions that operate on C-style strings. For this assignment, you will provide your own implementations of functions similar to two of these. Your solutions must conform exactly to the descriptions below, and your code must be organized exactly as described below.

```
/*********************************************************************
stringlength() returns the number of characters in the zero-terminated
C-string pointed to by the parameter s (zero if s is NULL).

If the target of s is not a zero-terminated C-string, the behavior of
the function is undefined (but probably annoying).

Pre:  s is NULL or points to a zero-terminated C-string

*********************************************************************/
uint32_t stringlength(const char* s);


/*********************************************************************
stringcopy() copies the string s to a new string, and returns a
pointer to the new string (i.e., to byte 0 of that array).

If the target of s is not a zero-terminated C-string, the behavior of
the function is undefined (but probably annoying).

Pre:  s is NULL or points to a zero-terminated C-string

*********************************************************************/
char* stringcopy(const char* s);
```

In the spirit of C, you must use pointer operations within your function implementations, rather than array notation. Recall how pointer arithmetic works in C. If p is a pointer to a cell in an array, then incrementing p will cause p to point to the next cell of the array (if there is one). So, the following code would print the contents of an array of integers A.

```
int *p = &A[0];
int i;

for (i = <number of values in A>; i < <number of values in A>; ++i) {

   printf("%d:   %d\n", i, *p);
   ++p;
}
```

Development using separate compilation:

For this assignment, you should follow common C practice to organize your solution and test code. You should produce three (or more) source files. One should contain your testing code, and can be called whatever you like. A sample file, `test.c` is provided on the course website.

The other two files will contain the declarations of your two functions, and any related `include` directives, and the actual implementations of your functions. The first file should be called `StrFns.h`; this is a C *header file*, and its purpose is to publish declarations that are to be used elsewhere in the program. You are already accustomed to using `include` directives to import portions of the C Standard Library into a program; the idea is the same, but now it's all your own code.

For this assignment, the header file should look like:

```
#ifndef STRFNS_H
#define STRFNS_H
#include <stdint.h>

uint32_t stringlength(const char* s);
char*    stringcopy(const char* s);

#endif
```

Note that the posted `test.c` file has an `include` directive for this header file. The effect of that is that the C pre-processor will copy the contents of the header file into `test.c` before the compiler begins to process the code, and therefore the compiler will find the declarations of the two functions in scope, making calls to them legal.

The first two lines in the header file, and the final line, are examples of *pre-processor directives*. You've already seen `include` directives; these serve a different purpose. The first directive, `ifndef`, begins a logical block that is terminated by the matching `endif` directive. The `ifndef` directive is followed by a symbol (usually a capitalized form of the name of the header file, but it could be something else). The effect is that if the symbol (`STRFNS_H` in this case) has already been defined, the pre-processor will omit everything from the `ifndef` until the matching `endif`. The second line is a `define` directive, that defines that same symbol.

The effect of all this is that the contents of the header file will never be imported twice into some source file, even if that source file has two different `include` directives for the header file. That's important because it's an error to have more than one declaration for the same name (like a function declaration) within the same scope. This is usually called the *multiple-inclusion problem*, and it is completely solved by using the pre-processor directives as shown above.

Whenever you write a C/C++ header file, you should always encase the body of it in a `ifndef-define-endif` construct as shown here. It may seem as though it would be very unlikely that the same header file might be imported more than once, but it's easier than you think, especially with large programs. All of the header files for the C Standard Library are protected in this manner.

The final file should be called `StrFns.c`, and it will contain the implementations of your functions, and any `include` directives needed for that code. As a general rule, you should not put `include` directives in a header file unless they are absolutely necessary there.

This demonstrates the common code-organization pattern used in C. Typically, collections of related functions are implemented in a common file, say `Foo.c`, and the declarations of some or all of those functions are placed in a corresponding header file, say `Foo.h`. If you have "private" functions that support other functions, but that the user doesn't need to call directly, you would just not put declarations for those functions into the header file, rendering them invisible to the client code.

What to turn in and how:

This assignment will be auto-graded using a test harness on the Curator system. The testing will be done under Windows XP using g++ 4.3.x.

Submit a single C source file containing your implementations of the two functions, and any necessary `include` directives, to the Curator System. Submit only the C function implementations and needed directives. Submit nothing else. Your solution should not write anything to standard output. Be sure that your header file only contains `include` directives for Standard C header files; any other `include` directives will certainly result in compilation errors.

Your submitted source file will be compiled with a test harness and a copy of the header file shown above, and the resulting output will be compared with correct output. The TA will also examine your code to determine if you've followed good C practice in your implementation.

Instructions, and the appropriate link, for submitting to the Curator are given in the *Student Guide* at the Curator website:

`http://www.cs.vt.edu/curator/`.

You will be allowed to submit your solution multiple times; the highest score will be counted.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the Programming Standards page in one of your submitted files.