

Secure Coding (CNT 4419) Assignment I

Objectives: To become acquainted with a C safe-string library and use it to implement basic string operations.

Due Date: Tuesday, March 24, 2026 at 11:59pm. No late submissions will be accepted.

Assignment Description

Complete this assignment by yourself.

While doing this assignment you will need to run a C compiler. One option is to use an online C compiler such as <https://paiza.io/en/languages/online-c-compiler>. If you use Paiza, you may optionally create an account to save your code. Paiza has an Input tab for providing input to the program.

This assignment asks you to use the Better Strings Library. An example program (*Main.c*) using this library can be found at <https://paiza.io/projects/OwRf7NzAYfmP5VDfpcMr6A>. This example includes sample input in the Input tab. You may fork this code (that is, add a copy to your account) using the menu next to the Run button.

If you are not using Paiza, begin by downloading the *bstrlib.h* and *bstrlib.c* files from the repository at <https://github.com/websnarf/bstrlib>. Then copy-paste the code shown in *Main.c* at the Paiza link above into your own *Main.c* file.

As part of the Better Strings Library, you will be using the `bgets`, `bdestroy`, `bconcat`, `bstrcmp`, and `binstr` functions, which are declared and implemented in the *bstrlib.h* and *bstrlib.c* files. These functions manipulate values of type `bstring`, a “better” string type (i.e., safer than the normal C-string type). Values of type `bstring` have two fields important for this assignment: `data` and `slen`. Field `data` refers to the underlying string (of type `unsigned char *`), and field `slen` refers to the string’s length (of type `int`). Your program can access these fields directly.

Below is documentation for library functions you may want to use. This documentation appears in the *bstrlib.txt* file in the GitHub repository.

bgets

```
extern bstring bgets (bNgetc getcPtr, void * parm, char terminator);
typedef int (* bNgetc) (void * parm);
```

Read a `bstring` from a stream. As many bytes as is necessary are read until the terminator is consumed or no more characters are available from the stream. If read from the stream, the terminator character will be appended to the end of the returned `bstring`. The `getcPtr` function must have the same semantics as the `fgetc` C library function (i.e., returning an integer whose value is negative when there are no more characters available, otherwise the value of the next available unsigned character from the stream.) The intention is that `parm` would contain the stream data context/state required (similar to the role of the `FILE*` I/O stream parameter of `fgetc`.) If no characters are read, or there is some other detectable error, `NULL` is returned.

bgets will never call the getcPtr function more often than necessary to construct its output (including a single call, if required, to determine that the stream contains no more characters.)

Abstracting the character stream function and terminator character allows for different stream devices and string formats other than '\n' terminated lines in a file if desired (consider \032 terminated email messages, in a UNIX mailbox for example.)

For files, this function can be used analogously as fgets as follows:

```
fp = fopen ( ... );  
if (fp) b = bgets ((bNgetc) fgetc, fp, '\n');
```

(Note that only one terminator character can be used, and that '\0' is not assumed to terminate the stream in addition to the terminator character. This is consistent with the semantics of fgets.)

[Hence,

```
    bstring b = bgets ((bNgetc) fgetc, stdin, '\n');  
reads a “better string” from stdin.]
```

bdestroy

```
extern int bdestroy (bstring b);
```

Deallocate the bstring passed. Passing NULL in as a parameter will have no effect. Note that both the header and the data portion of the bstring will be freed. No other bstring function which modifies one of its parameters will free or reallocate the header. Because of this, in general, bdestroy cannot be called on any declared struct tagbstring even if it is not write protected. A bstring which is write protected cannot be destroyed via the bdestroy call. Any attempt to do so will result in no action taken, and BSTR_ERR will be returned.

bconcat

```
extern int bconcat (bstring b0, const_bstring b1);
```

Concatenate the bstring b1 to the end of bstring b0. The value BSTR_OK is returned if the operation is successful, otherwise BSTR_ERR is returned.

binstr

```
extern int binstr (const_bstring s1, int pos, const_bstring s2);
```

Search for the bstring s2 in s1 starting at position pos and looking in a forward (increasing) direction. If it is found then it returns with the first position after pos where it is found, otherwise it returns BSTR_ERR. The algorithm used is brute force; $O(m*n)$.

bstrcmp

```
extern int bstrcmp (const_bstring b0, const_bstring b1);
```

Compare the bstrings b0 and b1 for ordering. If there is an error, SHRT_MIN is returned, otherwise a value less than or greater than zero, indicating that the bstring pointed to by b0 is lexicographically less than or greater than the bstring pointed to by b1 is returned. If the bstring

lengths are unequal but the characters up until the length of the shorter are equal then a value less than, or greater than zero, indicating that the bstring pointed to by `b0` is shorter or longer than the bstring pointed to by `b1` is returned. 0 is returned if and only if the two bstrings are the same. If the length of the bstrings are different, this function is $O(n)$. Like its standard C library counter part, the comparison does not proceed past any `'\0'` termination characters encountered.

Let $|s|$ denote the length of a string `s`. Use the `bstring` library to implement the following tasks.

1. Read a string `s`, print `s`, and print $|s|$ and whether $|s|$ is no greater than 5.
2. Read strings `s1` and `s2`, append `s1` to `s2` $|s1|$ times, and print the result.
3. Read strings `s3` and `s4` and print whether `s4` is a substring of `s3`.
4. Print the length of the longest common substring between `s3` and `s4`.
5. Print the input strings in lexicographic order.

Most of Task 1 has already been implemented for you in the given `Main.c` file, but you still need to add a bit of code to print the length of the input string and whether it is no greater than 5. Note that the given code calls `rmNewLine` to remove any final newline character from the input string and calls `bdestroy` to deallocate `bstring` memory. Call `rmNewLine` and `bdestroy` as appropriate throughout this assignment.

Submit to Canvas your completed `Main.c` file that performs the required tasks.

Notes:

- For this assignment, only the `Main.c`, `bstringlib.h` and `bstringlib.c` files are needed. Your `Main.c` must include only the `bstringlib.h` and standard-I/O (`stdio.h`) header files.
- Only strings input by the user need to be handled by the Better Strings library. Static strings may be character arrays, as seen in the example `Main.c` you have been provided.
- For full credit, your code must compile and execute without warnings or errors.
- Your code will be graded in part on readability, simplicity, and efficiency (time performance).

Sample Execution

Sample Input:

```
Input for Assignment 1
Code
Sec
Secure
S5
```

Sample Output (note the desired spacing on each line)

```
The first given input string is 'Input for Assignment 1'.
The length of this string is 22, which is greater than 5.
Appending ' Code' to 'Sec' 5 times: 'Sec Code Code Code Code Code'.
'S5' is not a substring of 'Secure'.
The longest common substring of 'Secure' and 'S5' has length 1.
Input strings in lexicographic order:
Code
Input for Assignment 1
S5
Sec
Secure
```

Optional Further Reading

The bstring library provides the following alternative means of accessing the data and slen fields.

char * bdata (bstring b);

Returns the char * data portion of the bstring b. If b is NULL, NULL is returned.

int blength (const_bstring b);

Returns the length of the bstring. If the bstring is NULL, the length returned is 0.

These accessors are documented in the *bstringlib.txt* file in the GitHub repository.

However, above this documentation, the following notice appears.

The macros described below are shown in a prototype form indicating their intended usage. Note that the parameters passed to these macros will be referenced multiple times. As with all macros, programmer care is required to guard against unintended side effects.

Macros are another source of security vulnerabilities in C. Details can be found in Chapter 2 of *The CERT C Secure Coding Standard* by Robert C. Seacord. (Reading from this book is optional and not required for this course.)