Dynamic Length and Static Length Strings in C, C#, Java, and Scala

Team "Who Knows Ada" : Nicholas Calkins, Erin Chon, Alan Huang, Harvey Lin
GitHub: https://github.com/sumoerin/cs_4080_WhoKnowsAda

I. Abstract

Our team sought to evaluate the question, "how are strings implemented in C, C#, Java, and Scala, and what are the costs and benefits associated with mutable dynamic length strings versus mutable static length strings?"

To answer the question of string implementation, each language was researched and findings were reported. To evaluate the costs and benefits in a reproducible and scholarly manner, we implemented the functionality of GNU ed in each of the four languages and ran experiments on inputs of certain lengths.

In the aggregate, we found that static length strings took more overhead in memory allocation compared to dynamic length strings, but when it came to speed the two were statistically the same. For the purposes of mutable strings in applications with constantly changing strings, our team would recommend using a buffer of dynamic length strings due to their definite memory efficiency and possible speed efficiency over static length strings.

II. Introduction/Concepts

A. Point/Objective of Project

Strings are an incredibly important aspect of modern programming languages, and much of what makes computers usable by humans. Our group was interested in the human computer interaction aspect of strings, so we opted for determining whether or not strings should be dynamic length or static length.

As we move further and further into the information age economy and trillions of bytes of data are sent and received each day, it is imperative to understand strings and the most optimal design thereof. Therefore, our group sought this topic as something not only important to the field of computer science but the very future of data itself.

B. Strings

A string is a linear sequence of characters or other points of data [1]. The characters in question can be alphanumeric or otherwise. The importance of strings in programming languages cannot be understated because as human beings, the way we understand is often in strings, which we might call "words" and "sentences."

C. Static Length Strings

A static length string is a string in which the length is set when the string is created [1]. The amount of characters in this string does not necessarily have to equal the length that was set, but it cannot be greater than the length that was set when it was created. This is much like the String class found in C#, Java, and Scala, except that our static length strings were mutable, unlike the strings in these classes which are static but immutable.

In the case of a concatenation or edit of a static length string, the operation would either not be done or be done as accurately as possible by truncating the string to the appropriate length.

D. Dynamic Length Strings

A dynamic length string is a string in which the string may be instantiated with some length, but it is allowed to grow with no maximum [1]. This option is the most flexible but comes with the most overhead with having to have growing and shrinking storage.

Technically, there is no way to have a truly dynamic length string, as there is a finite amount of storage space in the universe. As such, technically all dynamic length strings are limited length dynamic length strings. However, in common Computer Science vernacular, it is appropriate to call a string "dynamic length" as long as the string can grow without bound, potentially even to crash the program by overflowing the heap or stack, depending on where this dynamic length string is allocated.

E. How String Is Implemented in C

In C, there is no language provided string type. This design choice is likely due to the influence of B, the language's predecessor. Being an untyped language, all values in B are interpreted as either integer literals or memory addresses [8]. Consequently, the only language provided types in C

are literal values and pointers. Strings are instead represented as character arrays with the convention that they are always terminated with the null character '\0' and common string operations are provided in the string.h header file. If the character array used is declared as an array of constant character values, the string is immutable. Else, it is immutable.

F. How String is Implemented in C#

In C#, the string data type is used to store alphanumeric characters and is used to represent text. The String data type belongs to the system namespace in C#.net and its maximum size is the max value of int32, which is about 2147483647 characters. The C# string object is immutable, so it can't be changed once it is created [7]. To implement mutable static and dynamic length strings, we used the various overloaded constructors of the StringBuilder class that allows the user to specify both its pre-allocated length and its maximum allocated length.

G. How String is Implemented in Java/Scala

Strings in Scala are, for all intents and purposes, strings in Java. Scala is closely related to Java, given that Scala code is compiled into Java Virtual Machine bytecode. Scala strings are therefore just an alias for Java Strings [5]. For the intents of understanding string implementation in Scala, it is sufficient to understand the string in Java.

The Java String class is an immutable character sequence. As a result of the immutability, the String class in Java has a static length. The reason for this is made in an interview with James Gosling, the creator of the Java programming language and current advisor at LightBend, the company behind the Scala programming language [6].

In an interview with James Gosling conducted by Bill Venners, Venners poses the question, "When should we use immutables versus non-immutables?" The answer to this question gives great insight into the reasoning of making strings immutable in Java and Scala.

Gosling notes that the major reason why strings are immutable is because of the concept of "ownership" [6]. Strings are passed to open a file or sent to some API. With mutable strings, there is worry whether or not the string can be changed in a way that will break the program. With immutable strings, the receivers of the string object can be assured that the string will not change.

This ties into the other reason that Gosling gives, which is security. Gosling describes an attack where a string is mutated between a security check and an OS call [6]. Actions like this could allow malicious action, all because of a mutable string. This attack does not work with an immutable string.

Finally, strings are implemented as static in Java and Scala because it allows for potential benefits. Because strings should be immutable for security reasons, it also opens up an avenue for optimization. Gosling states that if an object is final and their fields are final, then this object can have a pure stack lifetime [6]. This could make for easier replication of strings, as stack lifetime is why primitives are put on the stack. For all intents and purposes, an immutable string implementation could give it the power of primitives in Java and Scala.

III. Methods/Algorithms

III.I GNU ed: Definition and Implementation

A. GNU ed, A Text Editor

GNU ed is a line-oriented text editor used to create, display, modify, and otherwise manipulate text files [3]. It works by having the user edit a buffer and invoking various commands to change the buffer. This buffer can be populated from an existing text file or written out to a text file, or both. It provides functionality for many operations on the text, but we will outline the ones that we included in our project. Notable features that we chose not to implement are support for a kill buffer, regular expression pattern matching, and command suffixing as they were not as relevant for our topic.

For testing the performance difference between static length and dynamic length strings, our group settled on implementing several commands and features of ed. Therefore, to understand how to run our code and understand our testing, there are some aspects that must be understood about ed.

B. Line Addressing and Special Addresses

GNU ed is predicated upon addresses, which is a line in the buffer. The first line is known as line 1, and continues onto line n, where n is the number of lines in the buffer. Addresses are given to commands which operate on those lines according to their specification. Special addresses are covered in the GNU ed manual but may be used in place of numerical addressing [3].

C. Commands in Ed

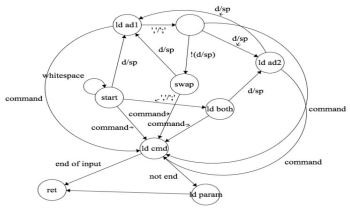
We implemented many ed commands in our own interpretation of ed. They are documented in ed's manual and therefore there will be only a brief overview of the commands we have implemented [3]. In our project we implemented functionality for appending, changing, deleting, copying, joining, and moving text, as well as reading from file and writing to file. We considered these commands to be sufficient for the purposes of our project. For example, here are three commands and their interpretations:

```
(1) 5a
(2) 1,3m7
(3) 45,60w myfile.txt
```

These commands are valid ed commands. (1) will allow the user to type in input and put that input under line 5, (2) will take lines 1 through 3 and move them to the lines under line 7, and (3) will write lines 45 to 60 out to a file called myfile.txt.

D. Parsing User Commands

Depending on the developer, this was implemented differently. The end goal of parsing the user input is to extract the addresses of the command, as well as the command itself. Optionally, getting a filename from the user input if the command desires one. Parsing also should account for bad addresses as well as nonsensical input. To aid in the process of parsing, this state machine of reading user input was developed by one of our team members.



Parses the format [address[(,/;)address]]command[parameter]

ld = load; ad = address; cmd = command; param = parameter

load the next character on each transition

d = character is a numerical digit

sp = character represents a special address value (excluding ',' and ';')

','/';' can either define both addresses or a single address so they are handled independant of the other special characters

*This parser only extracts the appropriate values for address1, address2, and parameter. error checking is left for after the input string is parsed

Implementing Static and Dynamic Strings in the Various Languages

E.1 Implementing Static Length Strings in C

Static length strings are naturally implemented in C through character arrays. Since the only restriction is that the allocated length for the string must not change during run-time, static length strings can be stack allocated as is the case with normal array declarations, heap allocated with malloc(), or even statically allocated if it is a string literal [9].

E.2. Implementing Dynamic Length Strings in C

Because the length of dynamic length strings is subject to change during run-time, they must be heap allocated. In C, such strings can easily be implemented using the built-in heap management functions malloc() and realloc() combined with character pointers.

E.3. Implementing GNU ed in C

As the emphasis of our design is on simplicity, our implementation of GNU ed in C is little more than a parser, a text buffer ADT, and a main driver.

The parser for this version is an implementation of the finite-state machine shown earlier and makes transitions for every character in the user-inputted line. This ensures that the parser is as efficient as possible for the number of transitions needed to parse a complete line of user input will never exceed the length of the line and no additional manipulation of the line (eg. splitting) is required.

The text buffer ADT utilizes a double linked list to store the buffer of text being edited and provides implementations for all the supported commands in the editor. Because C does not provide data structures in its standard library, a double linked list implementation was made with characteristics specific to this project. Two implementations of the nodes for the linked list were made, one using static length strings and one using dynamic length strings, so that the string type being used can be swapped with relative ease. The static string length node simply wrapped around a static length character array. For the dynamic length string node, a new user-defined type DynamicString was implemented and used. It is little more than a wrapper around a heap allocated character array created using malloc() and resized with realloc() when needed.

The main driver calls the relevant function from the text buffer ADT using a switch statement depending on the command and arguments extracted from the user-inputted line by the parser. The driver

is also responsible for entering and exiting input mode as well as error handling for when invalid or insufficient arguments are extracted.

For full details regarding the implementation, reference the source code available on the git repository.

F.1. Implementing Static Length Strings in C#

For our project, mutable static length strings were implemented using the StringBuilder class by setting the maximum allocated length to be equal to the pre-allocated length.

F.2. Implementing Dynamic Length Strings in C#

For mutable dynamic length strings, the StringBuilder is given an appropriately small pre-allocated length and its maximum allocated length is left uncapped.

F.3. Implementing GNU ed in C#

Similar to the other implementations, a double linked list was used to store the individual lines of text in the buffer, a parser class was made to store and read user commands, and a text buffer class provides implementations for the supported commands. Unaware of the standard library double linked list provided in C#, our groupmate decided to write a much less efficient custom implementation that ended up introducing significant overhead to the program. The main driver's behavior is similar to the other implementations. It merely uses the parser to extract arguments from user inputted lines and calls the relevant behavior of the text buffer class if the arguments are valid.

G.1. Implementing Static Length Strings in Java

String length of the String class in Java is naturally implemented as static. While just using the String class would be good for demonstrating static length strings, in order to implement the GNU ed, the strings need to be mutable as well, which strings in Java are not. In order to do so, the StringBuilder class is used instead. The StringBuilder class is similar to the String class in that they are both a sequence of characters stored in an array, but methods in the StringBuilder class allow for variable length and elements in the array. One of the main differences between the String class and the StringBuilder class is that StringBuilders have a capacity. Limiting this capacity can simulate a static length string, while also being mutable.

G.2. Implementing Dynamic Length Strings in Java

Because string length is naturally static for Java, the StringBuilder class is also used to implement dynamic length strings. This is much simpler than implementing static length strings as the StringBuilder class itself already has a dynamic length string implementation. Instead of limiting the capacity of the string builder, the string is free to grow or shrink during runtime beyond its initial capacity.

G.3. Implementing GNU ed in Java

Implementing ed in Java starts with making the text buffer, and two variants: one with dynamic length mutable strings and one with static length mutable strings. The most technically demanding portion is the parser, much like the other implementations.

Following good encapsulation, the parser is made its own class. It follows the logic of our state machine. The main is then made, which has a text buffer and a parser, and its job is to take input from the user. From here, it's just a matter of checking the user input to see if it's doable, like making sure the addresses and commands are valid.

H.1. Implementing Static Length Strings in Scala

By utilizing the StringBuilder class, it is possible to make a mutable string with a static length. While StringBuilder is a Java class, Scala is designed to use Java classes because Scala compiles in JVM bytecode and runs on the JVM. Therefore, StringBuilder is a fine basis for implementing a static length string in Scala.

StringBuilder can be instantiated with a capacity, which is the size of the initial value array in the StringBuilder. When the StringBuilder is constructed, it is constructed with a capacity desired by the user. Then, an immutable String object is appended to this empty StringBuilder. The StringBuilder isn't growing or shrinking because the length of the characters is less than the capacity. If an operation on this static length string would ever go beyond the max capacity as defined by the user, then the operation would be refused.

H.2. Implementing Dynamic Length Strings in Scala

By utilizing the StringBuilder class once again, it is possible to make a mutable string with a dynamic length. StringBuilders are already a dynamic length, mutable string, having already an initial capacity that grows as the various operations on the string extend it past the capacity. Therefore, implementing dynamic length strings in Scala is as simple as using the StringBuilder class with no restrictions on size.

H.3. Implementing GNU ed in Scala

Implementing GNU ed in Scala was an iterative process where we started with a general buffer then developed a dynamic length string buffer and then a static length string buffer. We finally made a driver for this code where the parser parses the user commands and accomplish them if they are valid.

H.3.1 The Dynamic Length String Buffer (DynamicBuffer)

We started development by designing a buffer. This buffer had an internal ArrayBuffer that took types of StringBuilder. We implemented several functionalities that would be needed later for ed.

We implemented several functionalities to have StringBuilders put into the ArrayBuffer. insertAtLoc and insertBeforeLoc took an ArrayBuffer of String objects and had them put into the main buffer. We also implemented functionality to read String objects from a file. These String objects were converted to StringBuilders by a helper method, getStringBuilders, which returned an ArrayBuffer of StringBuilders.

From there we implemented functionality to join two lines in the buffer together. We also added a function to delete a line from the buffer, as well as functions to print a line in the buffer. Finally we created functionality to get the size and last element location in the buffer.

H.3.2 The Static Length String Buffer (StaticBuffer)

The Static Length String Buffer is an extension of the Dynamic Buffer. This intuitively made sense as well, given that a “static” StringBuilder is just one that is limited. Therefore, we overrode several methods in the DynamicBuffer class to make this a true static length string buffer.

For one, there is a new value associated with the buffer which is called maxLength. This variable can be edited by the user to indicate the initial and max capacity of any StringBuilder in the buffer.

The joinLines method was overwritten to ensure that no two lines could be joined if their new length would exceed the maxLength.

Append, insertAtLoc, insertBeforeLoc, and readFromFile were all overridden to check the Strings coming in, to make sure that they could actually be inserted. If they were longer than maxLength, they would not be added to the buffer. Otherwise, they would be converted into a new StringBuilder, allocated with the values of the String object and a capacity of maxLength.

H.3.3 Driving the Buffer (BufferCommands)

First, we implemented the commands of ed. These are the ones mentioned in III.I.C, and can be seen in my source code or referenced from there. These methods operate directly on the buffer and use the commands delineated there. There is some level of address checking as well in these methods, for example when moving lines, the destination can’t be in the range of the addresses.

Also, we made the parser for the user input. The user enters their input in and we turn it into a StringBuilder, due to the immutable properties. The parsing takes place in many helper methods but is driven in main.

First, the parser checks for an address. If it finds one, it deletes the address out of the user input and looks for a comma. If the program doesn’t find one, set the second address to a “null” value of -1. Then, the program looks for a command. If there isn’t one, then the program will simply print the address line if it exists. Finally, knowing the command, the program looks for a file name and uses it if it finds it, otherwise uses the default filename of the user.

IV. Experiments

A. Experiments Run in C

To assess the performance impact of the two types of strings, the times taken to insert a predetermined set of text when using a static length string implementation and a dynamic length string implementation were measured and compared. The editor was invoked 10,000 times with 25 lines of approximately 200 characters inserted and lines past line 9 deleted every time. The specific script timed is available on the git repository as “timethis.sh”. To eliminate variance due to resource competition by other programs, the CPU time was measured instead of the real time. For this test, a static length 256 characters and an initial dynamic string length of 10

characters were used. These are the observed CPU times over 5 trials.

Static Length Strings

Trial 1: 18.86s
Trial 2: 18.98s
Trial 3: 18.79s
Trial 4: 19.33s
Trial 5: 18.87s

Dynamic Length Strings

Trial 1: 19.12s
Trial 2: 18.86s
Trial 3: 20.68s
Trial 4: 19.35s
Trial 5: 18.90s

To measure the memory impact of the two string implementations, a predetermined body of approximately 130,000 lines of biblical text no longer than 512 characters were read into the editor. The RAM usage of the program was then measured using the provided memory inspector in the IDE Xcode for both the static length and dynamic length string implementations. For this test, a static string length of 1024 characters and an initial dynamic string length of 8 characters were used. For C, the memory usage did not fluctuate from trial to trial so only one measurement is given. These are the observed memory usages using the two string types.

Static Length Strings: 189.1MB
Dynamic Length Strings: 41.2MB

B. Experiments Run in C#

Similar to the tests run for the C implementation, approximately 130,000 lines of biblical text was loaded into the C# implementation of the editor with a static length of 512 and an initial dynamic length of 10. Due to the comparably large amount of time this implementation required to load the lines of text, the stopwatch class was used to measure the load time for all 130,000 lines in order to perform a time comparison. Visual Studio's diagnostic table was used to monitor the memory usage between the two string types.

Static Length Strings

Trial 1: 6.26 min
Trial 2: 6.07 min
Trial 3: 7.65 min
Trial 4: 6.47 min
Trial 5: 6.47 min

Dynamic Length Strings

Trial 1: 6.25 min
Trial 2: 6.06 min
Trial 3: 5.47 min
Trial 4: 5.11 min
Trial 5: 6.92 min

Static Length Strings: 344 MB
Dynamic Length Strings: 96 MB

C. Experiments Run in Java

To gauge memory usage and time taken depending on the implementation of string, a body of text was written into the buffer approximating 130,000 lines. There was no write out of this text, because Java took too long to do that. This test was run 30 times per buffer implementation, and the memory usage and time taken was recorded. This was done using Java's Runtime library and a python script, respectively. Given the remarkably low deviation between each of the 30 trials, it is not necessary to show the histograms of the data but instead the averages will be given.

In 30 trials, the average memory usage and speed of the two buffers is shown below:

Dynamic Length Strings: 36.7 MB
Static Length Strings: 139.6 MB

Dynamic Length Strings: 0.31 seconds
Static Length Strings: 0.0402 seconds

D. Experiments Run in Scala

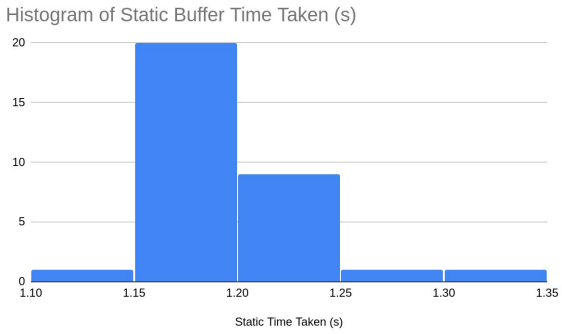
To gauge the memory usage and time taken for each different backend, a body of text was appended to each buffer before being written into a file. This body of text was approximately 130,000 lines and had no line greater than 512 characters. A length of 512 was used for the maxLength of the static StringBuilders.

This test was run 32 times per buffer. The test was timed using a Python script which calculated the elapsed time the test took. The memory usage was calculated using Java's java.lang.Runtime, which is

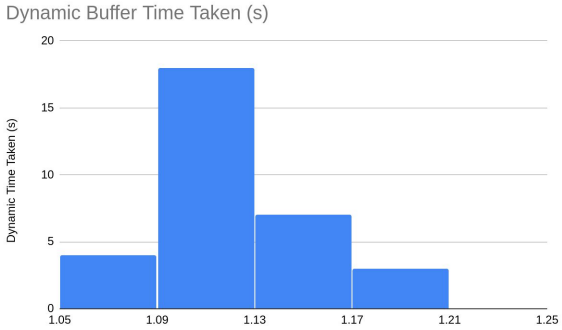
used to calculate the used memory by subtracting total memory from free memory.

D.1. Speed Differences Between String Lengths

The raw data of the experiment is made available with this paper. Therefore, we will show charts from the data and make inferences and speculations thereof.



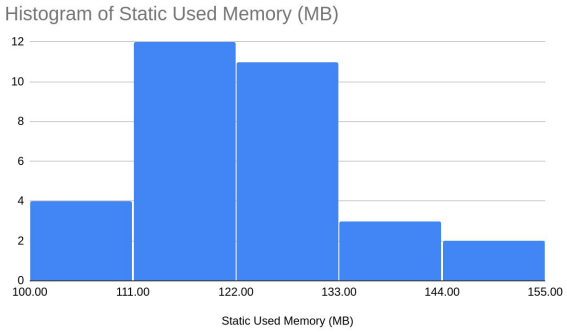
This is the histogram of the speeds of the various trials of inserting and writing out the static length string buffer. The speed settled on an average of 1.20 per trial, with very low spread overall.



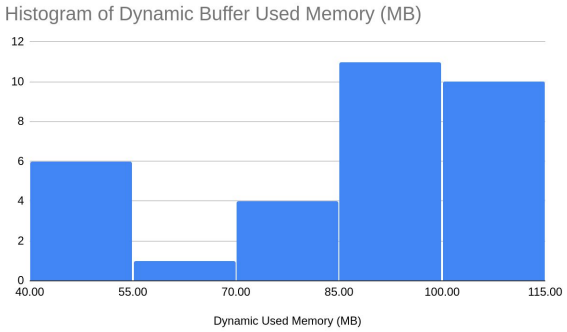
This is the histogram of the speeds of the various trials of the dynamic length string buffer. The average speed is 1.12, which we find to be significantly greater than that of the static length string implementation.

D.2. Memory Usage Differences Between String Lengths

The raw data of the experiment is made available with this paper. Therefore we will show charts from the data and make inferences and speculations thereof.



This is the histogram of the memory usages of the various trials of inserting and writing out the static length string buffer. The memory usage settled on an average of 124.5 MB per trial. The standard deviation of these trials was 11.3 MB.



This is the histogram of the memory usages of the various trials of inserting and writing out the dynamic length string buffer. The memory usage settled on an average of 85.5 MB per trial. The standard deviation of these trials was 21 MB.

V. Results

A. Results of Experiments in C

For C, the performance impact of static length strings over dynamic length strings was small but notable. Averaged over 5 trials, static length strings provided an approximately 2% increase in speed over dynamic length strings. Static length strings are expected to be faster as dynamic length strings experienced the overhead of heap allocation, reallocation, and deallocation in this test. However, the difference is surprisingly small. This can likely be attributed to the efficiency of the realloc() function in C or the negligible overhead of heap allocation in general for data of this size.

For memory usage, dynamic length strings provided a significant decrease in RAM usage. While this difference will vary depending on the default lengths of both string implementations and the

average lengths of the lines of text read in, dynamic strings used 79.2% less RAM for our test case when compared to static length strings.

B. Results of Experiments in C#

For C# the dynamic length string loading time was about 10.94% faster than that of the static length string implementation. Memory usage for static length strings was about 112.73% more than the dynamic length string. From the visual studio memory usage graph, we can see the heap garbage collection occurs 7 times for the dynamic length string, and 22 times for the static length string.

C. Results of Experiments in Java

Much like the results of other experiments, dynamic length strings won out entirely when it came to memory. In general, our static length string buffer implementation used 300% more memory than the dynamic implementation! This gap is quite large but expected given the results in other languages.

In terms of speed, the dynamic implementation is also markedly faster, clocking in at about 33% faster than the static length implementation. This, much like C# and Scala, can be attributed to the overhead of not allowing the String to grow beyond a certain size.

D. Results of Experiments in Scala

We surmise that the difference in speed can be attributed to the fact that the dynamic length strings on average have a lower allocation time, because the average length in bigbible.txt is less than 512. The static buffer, on the other hand, must ensure that every string has a capacity of 512, regardless of whether or not the string actually has that many characters.

We surmise the difference in memory usage can be attributed to the difference in allocation size. As mentioned in the speed tests, the lines of bigbible.txt don't typically approximate 512 characters, and as a result the static length string buffer will always be overallocating.

What is interesting to note is that the static string length memory usage is markedly more consistent than that of the dynamic length string buffer. Therefore, perhaps it might be more suitable to use mutable static length strings when mutable strings are needed but a consistent memory usage is more important than a low memory usage.

VI. Future work

In the future, our team would like to test and find the optimal way to use static strings. Static strings can be immutable, which is great for security and ensuring that the string stays faithful to the original user input. Although our group found that the dynamic length mutable string was vastly superior, the security and thread safety that comes from a static, immutable string cannot be overlooked and deserves its own due diligence.

Other than this, we would like to continue our work in developing a line text editor, finding the optimal way to implement the buffer and thinking of other commands that users might find useful. Despite the learning curve, there are many applications of a line based text editor, and these applications can be explored in the future.

VII. Conclusion

For C, it can be seen that the performance impact of dynamic length strings is very small while the potential gains in memory efficiency are large. Additionally, dynamic length strings are more reliable as they mitigate the issue of buffer overflow that plagues static length string implementations. Given this, it is clear why many languages provide native support for dynamic length string types. However, the exclusion of a dynamic length string type, and string types as a whole, from C is not surprising given its history. Considering the ease of implementing a user defined string type of both static and dynamic lengths using C's standard library headers, it is therefore not a great loss that C does not provide its own string type.

For C#, we can see that dynamic length strings are much more efficient than static length strings. The reason could be that static length strings require a higher memory allocation to create each line. Alternatively, the speed difference could be due to our groupmate's poor choice in parameter passing. For this implementation's linked list, our group member chose to pass a copy of the list's head node by value for each insertion operation. Since static length strings have a larger preallocated size, this naturally introduces much more overhead for static length strings.

For Scala, it follows the same logic of Java for why its native Strings are static and immutable. It's largely a security feature, coming at the cost of lack of flexibility. Programming languages must make sacrifices and the developers of Java and Scala

preferred the security of a static length and immutable String much more than the flexibility of a dynamic length and mutable String.

In the experimentation, it can be seen that dynamic length strings have a much lower memory allocation for static length strings. This has to do with the fact that the dynamic length string is able to grow, but the static length string must be allocated from the beginning, and the allocation length for the static length string may be in excess of what is needed.

Dynamic length strings also produced a faster runtime in the buffer, and this is due in part to the fact that static length strings need to check each incoming character sequence length to ensure it can hold it. Perhaps without this overhead, it can run as fast (if not faster) than the dynamic length string buffer implementation of ed.

In Java, we saw largely the same results we saw in C and C#. Memory usage was much lower for dynamic length strings than static length strings, and the dynamic length strings ran faster as well.

Overall, our group found that static length mutable strings use much more memory than dynamic length mutable strings. In terms of speed there was no clear winner, but in theory the allocation and deallocation of static length strings should be faster. It is worth noting that in safety critical and other security applications, strings should be immutable and therefore static length. For our application of a text editor where strings are constantly changing, we would recommend dynamic length mutable strings due to their lower memory usage.

REFERENCES

[1] R. W. Sebesta, *Concepts of Programming Languages*, 12th ed. New York , NY: Pearson, 2019.

[2]https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

[3]https://www.gnu.org/software/ed/manual/ed_manual.html

[4]http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/lang/AbstractStringBuilder.java

[5]https://www.scala-lang.org/api/current/scala/Predef$.html

[6]https://www.artima.com/intv/gosling3.html

[7]https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/

[8]http://www.bell-labs.com/usr/dmr/www/chist.html

[9]https://en.cppreference.com/w/cpp/language/string_literal

[10]https://docs.oracle.com/javase/tutorial/java/data/buffers.html

[11]http://cs.boisestate.edu/~alark/cs354/lectures/data_types_chars_strings.pdf