Outline

- In this lesson, we will:
  - Define strings
  - Describe how to use character arrays for C-style strings
  - Look at the length of strings
  - Consider string operations, specifically distances
  - Learn how to manipulate strings
  - Look at other alphabets and Unicode

Strings

- An array stores a list of values
  - E.g., temperatures, voltages, positions, speeds, etc.
- Generally, each value has independent significance
- An array of characters, however, has the following properties:
  - The significance comes from how the characters are strung together:

    post pots spot stop tops opts spto
  - The characters come from a small *alphabet*
- If the characters of an array come from a fixed *alphabet*, the array is called a *string of characters*, or simply a *string*
  - The alphabet for character arrays (C-style strings) is the set of all ASCII characters
  - More inclusive strings use Unicode
- The *length* of a string is the number of characters

String length

- C-style strings are defined as:
  - An array of characters where the entry following the last character is the null character `'\0'` with a value of `0x00`
    - `'\0'` must be included in the array length, but not the string length
    - The string length is at least one less than the array length
  - We will prefix all identifiers that are pointers to strings with `"s_"`
- We can determine the length of a string:

```
std::size_t string_length( char s_str[] ) {
    for ( std::size_t k{0}; true; ++k ) {
        if ( s_str[k] == '\0' ) {
            return k;
        }
    }
}
```

What happens if you forget the `'\0'`?

String length

- Important:
  - `'a'` is a single null character
  - `"a"` is an array occupying 2 bytes
    - The first entry is `'a'` and the second is `'\0'`
  - Oddly enough, `"\0"` is an array occupying 2 bytes
    - Both entries are `'\0'`

String length

- Suppose we have an argument:

s_str →

0	1	2	3	4	5	6	7	8	9	10	11
E	C	E	⍰	1	5	0	\0	?	?	?	?

  - Any characters after the `'\0'` are ignored

```
std::size_t string_length( char s_str[] ) {
    for ( std::size_t k{0}; true; ++k ) {
        if ( s_str[k] == '\0' ) {
            return k;
        }
    }
}
```


String length

- We initialize k to zero and step through the array

s_str →

0	1	2	3	4	5	6	7	8	9	10	11
E	C	E	⍰	1	5	0	\0	?	?	?	?

k: 0

```
std::size_t string_length( char *s_str ) {
    for ( std::size_t k{0}; true; ++k ) {
        if ( s_str[k] == '\0' ) {
            return k;
        }
    }
}
```


String length

- When we get to the null character, we return:

s_str →

0	1	2	3	4	5	6	7	8	9	10	11
E	C	E	⍰	1	5	0	\0	?	?	?	?

k: 7

```
std::size_t string_length( char *s_str ) {
    for ( std::size_t k{0}; true; ++k ) {
        if ( s_str[k] == '\0' ) {
            return k;
        }
    }
}
```

String length

- Question: What happens if you forget to include a null character?

s_str →

0	1	2	3	4	5	6	7	8	9	10	11
E	C	E	⍰	1	5	0	?	?	?	?	?

  – It will continue until it finds a '\0' (0x00) or it causes a segmentation fault

```
std::size_t string_length( char *s_str ) {
    for ( std::size_t k{0}; true; ++k ) {
        if ( s_str[k] == '\0' ) {
            return k;
        }
    }
}
```


Printing of strings

- The `std::cout` object treats character arrays as special:
  – It is assumed that if you are printing a character array, that array is a string

```
#include <iostream>

int main();

int main() {
    char s_hi[]{"Hello world!"};
    std::cout << s_hi << std::endl;
    std::cout << static_cast<void *>( s_hi )
              << std::endl;
    return 0;
}
```

Output:

```
Hello world!
0x7ffcbfbf75eb
```


Operations on strings

- There is a significant amount of work into strings
  – Extracting or finding substrings
  – Describing or finding patterns
    - Matching case or not
    - Defining whitespace and finding only *whole words*

Distances between strings

- One important question is how similar are two strings?
  – How *close* are two strings?
  – Consider:
    `"Et tu, Brute?"`
    `"t tu, Brute?"`
    `"Et ut, Brute?"`
    `"Et tu, Brune?"`
  – The *Levenshtein distance* is defined as the minimum number of *edits* required to convert one string to another
  – One edit is defined as
    - Inserting or removing a character
    - Replacing a character
    - Swapping two adjacent characters

Distances between strings

- For example, you could use the Levenshtein distance to determine which words to suggest in a spell checker
  - For example: "incomprehssible" is not a word, but

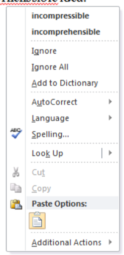
```
incompre~~h~~ssible        incomprehssible
incompressible          incomprehesible
                        incomprehensible
```

  - This word is:
    - One edit away from incompressible
    - Two edits away from incomprehensible
  - Recommend "incompressible" first…

Distances between strings

- What's wrong with this picture?

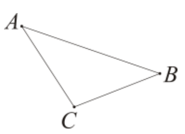
This is an incomprhenssible idea.



  - The distance is context insensitive
    - Ideas cannot be incompressible, so suggest the second first…

Distances between strings

- Recall the properties of the Euclidean distance:
  - dist(A, B) ≥ 0
  - dist(A, B) = 0 if and only if $A = B$
  - dist(A, B) = dist(B, A)
  - dist(A, B) ≤ dist(A, C) + dist(C, A)



- All of these properties hold for the Levenshtein distance between strings

Strings in other alphabets

- Other alphabets include:
  - Morse code uses five characters:
    - dot
    - dash
    - inter-character space
    - inter-word space
    - inter-sentence space
  - Note: "SOS" is · · · - - - · · · while the mayday $\overline{\text{SOS}}$ is · · · - - - · · ·

inter-character spaces

Strings in other alphabets

- Western European alphabets often include additional characters on top of ASCII; however, Unicode allows for most alphabets

German	ABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜß
Swedish	ABCDEFGHIJKLMNOPQRSTUVWXYZÅÄÖ
Italian	ABCDEFGHILMNOPRSTUVZ
Slovenian	ABCČDEFGHIJKLMNOPRSŠTUVZŽ
Polish	AĄBCĆDEĘFGHIJKLŁMNŃOÓPQRSŚTUWXYZŹŻ
Greek	ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ
Russian	АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩъыьЭЮЯ
Persian	ا ب پ ت ث ج چ ح خ د ذ ر ز ژ س ش ص ض ط ظ ع غ ف ق ک گ ل م ن و ه ی
Gurmukhi	ੳਅੲਸਹਕਖਗਘਙਚਛਜਝਞਟਠਡਢਣਤਥਦਧਨਪਫਬਭਮਯਰਲਵੜ

Strings in nature

- Even better, deoxyribonucleic acid (DNA) is a string with a four-characters alphabet:
  - cytosine C
  - guanine G
  - adenine A
  - thymine T

- All the algorithms developed by computer scientists for analyzing and manipulating strings were immediately transferable to the analysis and manipulation of DNA
  - This is one of the beauties of *abstraction*

Summary

- Following this lesson, you now
  - Know that strings are sequences of characters
    - Those characters come from a fixed alphabet
  - Know the most primitive means of storing strings are null-character-terminated arrays of `char`
  - Understand how to calculate the length of a string
  - Understand string distances
  - Are aware that
    - Simple strings are limited to ASCII
    - Other languages require Unicode

References

[1] No references?

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a s_result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.