String data types

- One of the most widely-supported data types, simply due to the amount of text interaction generally necessary with users
- String types may be explicitly implemented in a language, provided via libraries, or implicitly supported through arrays (or vectors) of characters

Implementation

- Most implementations implement strings as arrays (or vectors) of characters, even if that is hidden from developers
- Static vs dynamic length an issue: do we fix the size of a string in advance (and possibly use only part of it) or allow it to be resized during execution
- C uses the static approach, with a null terminator to indicate end of portion currently in use (alternative static approach is to store the string size in first byte(s) of string)
- Resizing and storage implementation issues will be considered in detail with arrays

Null-terminated strings

- The C convention (as followed by the string.h library routines) is to allocate a fixed size array of characters to hold the “string”, and mark the end of the active portion of that by the null (ascii 0) character
- String manipulation routines thus don’t care about the actual array size, they just process from any point in the string and continue to the null
- This means you cannot simply look up the size of a string, e.g. for copying or bounds checking, you have to search for the null character to compute the size.
- It also means that failing to store the null can result in string routines reading or writing memory far past the true end of the character array

Size-tracked strings

- Many languages store the current length of a string, and possibly also the amount of space allocated to it (e.g. right now we are using the first 37 bytes of 128 bytes allocated to string s)
- This requires extra storage fields, but allows for easier bounds checking
- This is a common approach for string classes in OO languages
- (again, we'll consider storage allocation and resizing when we get to arrays)

Typical operations

- Copying or overwriting the string (note the related bounds checking issues)
- Inserting content within the string
- Reading or writing one specific character by position
- Searching the string for instances of a character or pattern, often with the ability to delete or replace the targetted pattern with a new pattern
- Looking up the length of the string
- Converting between upper/lowercase
- Whitespace manipulation (trimming, padding, aligning, etc)

Strings of variable-size chars

- unicode can support variable-sized characters, with the character stored in the smallest number of bytes managable (e.g. 1, 2, 4, etc)
- different bit patterns are used for the first byte in a unicode string than for “continuation” bytes
- this means the byte length of an n-character string depends on the specific characters store, and the distance of the i’th character (in bytes) from the start of the string depends on which specific characters precede it

Impact on operations

- This means that to access the i’th character the compiler can’t simply compute an offset to that character, it actually needs to traverse the bytes in the string, counting characters as it goes
- This impacts the efficiency of many common string operations (access i’th character, substring operations, appends, etc)
- These become O(N) operations instead of O(1)
- Many languages choose a fixed number of bytes (e.g. 8) for unicode characters to eliminate this, at the cost of wasted storage space