CSCI 135
Strings

Strings

- A `char` literal is indicated with single quotes (*e.g.*, 'c'), while a `string` literal is indicated with double quotes (*e.g.*, "abc").
- These are C++ strings and are in <`string`> library They are incompatible with C strings (in <`cstring`> library).
- String operators include:
  - =: Assignment
  - $<, >, <=, >=, ==, !=, \ldots$: Comparison operators, using lexicographic ordering based on ASCII ordering.
  - `s[i]`: Indexing, *i.e.*, i'th character of s ($i \geq 0$)
    Ex: `string s = ``cat''; char c = s[2]` sets c to 't'
  - +: Concatenation (overloaded for characters)
    Ex: `` `apple'' + ``pear'' `` evaluates to "applepear"
    while `` `apple''+'s' `` evaluates to "apples"
  - `s.empty()`: true iff s is an empty string
  - `s.length()`: length of s (≥ 0)
  - Various other operators to access and manipulate strings

⚠Assignment and comparison operators work for strings but NOT for all C++ data types.

String Example

```
string age = "21"                          ≠ the int 21
if (age.length()==2)                       be careful!
{
   char decade = age[0];                   the character '2'
   intdecade = (int) (age[0]-'0');         the number 2
}
age = '2';                                 compile error!
char c = age[2];                           runtime error!
```

⚠Don't confuse 1-character strings with characters (or numeric characters with numbers).

⚠Always ensure that a string index is in [0...s.length()-1]

String I/O operators

- <<, >>: input/output (still uses whitespace as delimiter).
  e.g., `cin >> s1 >> s2` would assign "hi" to s1 and "mom" to s2, if the user typed "hi mom".
- `getline`: may input spaces, as before.
  e.g., `getline(cin,s1)` would assign "hi mom" to s1 in above example.
  (though its possible to control field delimiters in getline)

```
int n; string line;
cin >> n;
getline(cin,line);
```

- Input is "hello hitchiker" ⇒ error
- Input is "42 hello hitchhiker" ⇒ n gets 42; line gets "hello hitchiker"
- Input is "42 \n hello hitchhiker" (*i.e.*, "42" followed by enter followed by "hello hitchhiker") ⇒ n gets 42 and line gets the empty string (since the cin doesnt consume the newline character)
- Input is "42 57" ⇒ n gets 42 and line gets the *string* "57"

Example: Reverse a String

Spec: Input a string and display its reverse on the screen.

Start by writing pseudocode:

1. Prompt for and input string.
2. Initialize an index variable (`theIndex`) to the position of the last character of the string.
3. Iterate backwards, outputting the character currently indexed by `theIndex`.

Example: Reverse a String

Spec: Input a string and display its reverse on the screen.

Start by writing pseudocode:

1. Prompt for and input string.
2. Initialize an index variable (`theIndex`) to the position of the last character of the string.
3. Iterate backwards, outputting the character currently indexed by `theIndex`.

```
string aString;
cout << "Enter string to be reversed";
cin >> aString;
int theIndex = aString.length() - 1;
while (theIndex >= 0) {
   cout << aString[theIndex];
   theIndex--;
};
cout << '\n';
```

Example: Upper Case

Spec: Convert `someString` from lower to upper case

Observation (from ASCII table): The uppercase ASCII characters (65-90) are exactly 32 less than their corresponding lower case characters (97-122).

```
for (int i=0; i<someString.length(); i++)
  if ((someString[i]>='a') && (someString[i]<='z'))
    someString[i] = (char) ((int) someString[i] - 32);
```

Exercise for reader: This *mutates* `someString`. Rewrite this to store result in a new string (say `ucstr`) instead.

Example: 2nd Word in Sentence

Spec: Given a sentence `sent`, store the 2nd word in `word2`.

Def: A word is any space delimited string (possibly multiple spaces between words).

Precondition: `sent` is a nonempty string with at least 3 words and doesn't begin with a space.

Postcondition: `word2` is the 2nd word in `sent`.

```
// Iterate through sent from beginning until a space is found
// Postcondition: sent[pos]= ' ' and sent[i] != ' ' for i<pos

// Iterate through sent from sent[pos] until non-space is found
// Postcondition: sent[pos2] is the first non-space character
//   in sent after sent[pos]

// Iterate through sent from sent[pos2] until a space is found
// Postcond: sent[pos3] is first space in sent after sent[pos2]

// Copy characters from sent[pos2] to sent[pos3] into word2
// Postcondition: word2 is the 2nd word in sent.
```

2nd Word in Sentence - Filling In The Holes

```
// Precondition: sent is a string with at least 3 words
// Iterate through sent from beginning until a space is found
int pos,pos2,pos3=0;
while (sent[pos] != ' ') pos++;
// Postcondition: sent[pos]= ' ' and sent[i] != ' ' for i<pos
// Iterate through sent from sent[pos] until non-space is found
for (pos2=pos; sent[pos2] == ' '; pos2++);
   OK but probably better style to use same construct forall [similar] loops
// Postcondition: sent[pos2] is the first non-space character
//   in sent after sent[pos]
// Iterate through sent from sent[pos2] until a space is found
for (pos3=pos2; sent[pos3] != ' '; pos3++);
// Postcond: sent[pos3] is first space in sent after sent[pos2]
// Copy characters from sent[pos2] to sent[pos3] into word2
for (int i=pos2; i<pos3; i++)   i local to loop
  word2[i-pos2] = sent[i];
// Postcondition: word2 is the 2nd word in sent.
```

⚠Postcondition of each step is precondition of next step, and we only need to look at preious postconditions to figure out what to write next (though we have been a bit imprecise, since a postcondition also needs to include all earlier postconditions that are still applicable and needed).

2nd Word in Sentence - Neater Version

```
// Precondition: sent is a string with at least 3 words
int pos=0, pos2=0, pos3=0;
while (sent[pos] != ' ') pos++;
// Assert: sent[pos]= ' ' and sent[i] != ' ' for i<pos
for (pos2=pos; sent[pos2] == ' '; pos2++);
// Assert: sent[pos2] is the first non-space character
//   in sent after sent[pos]
for (pos3=pos2; sent[pos3] != ' '; pos3++);
// Assert: sent[pos3] is first space in sent after sent[pos2]
for (int i=pos2; i<pos3; i++)
  word2[i-pos2] = sent[i];
// Postcondition: word2 is the 2nd word in sent.
```

The use of asserts makes the code readable, maintainable, and [almost] self-documenting!

Example: Words starting with c (Pseudocode)

Spec: Given a sentence `sent` and a character c, print all words starting with c.

First try (pseudocode):

```
while (more characters exist in sent) {
  wordPos = start position of next word
  wordEndPos = position of word end
  if word at wordPos starts with c
    print word
  advance wordPos to after word
}
```

❓ What condition causes while loop to terminate? Do we need to keep track of any extra variables?

Words starting with c

```
wordPos = 0; wordEndPos = 0;
while (wordPos < sent.length()) {
   wordPos = findNextWord(sent,wordPos);
   wordEndPos = findWordEnd(sent,wordPos);
   if (sent[wordPos] == c)
      printWord(sent,wordPos,wordEndPos);
   wordPos = wordEndPos+1;
}
```

⚠ Don't think about *how* find* work when writing this

ⓘ Since findNextWord and findWordEnd both scan through sentence, maybe we can combine these into a more general function? Exercise for reader.

Words starting with c - 2

```
// Precond:
//   sent[ind] is whitespace (space or tab)
// Postcond:
//   The return value is the smallest k>ind such
//   that sent[k] is not a whitespace character
//   (or sent.length() if no such k exists)
int findNextWord(string sent, int ind) {
  for (int i=ind+1; i<sent.Length(); i++)
    if ((sent[i] != ' ') && (sent[i] != '\t'))
      return (i);
  return (sent.length());      no word found
}
```

Words starting with c - 3

```
// Precond: 0 <= ind < sent.length()
// Postcond:
//   The return value is the index of the last
//   character of the word starting at sent[ind]
int findWordEnd(string sent, int ind) {
  for (int i=ind; i<sent.length()-1; i++)
    if ((sent[i+1] == ' ') || (sent[i+1] == '\t')
      return (i);
  return (i);
```

⚠ Be careful on loop termination condition above!

Words starting with c - 4

```
// Precond: 0 <= sPos < sent.length(),
//          0 <= ePos < sent.length()
// Postcond: The sent substring between indices
//   sPos and ePos (inclusive) are printed,
//   followed by a space
void printWord(string sent, int sPos, int ePos) {
  for (int i=spos; i<=ePos; i++)
    cout << sent[i];
  cout << ' ';
}
```

Naive String Search

Precondition: s is a nonempty string, and p is a nonempty pattern string.
Postcondition: true is returned iff p is found in s.

```
int i;
for (i=0; i<=s.length()-p.length(); i++) {
  // assert: p does not start at positions 0..i-1 of s
  ...
}
return (i != s.length()-p.length());
```

? Now what?

Naive String Search

Precondition: s is a nonempty string, and p is a nonempty pattern string.
Postcondition: true is returned iff p is found in s.

```
for (int i=0; i<s.length()-p.length(); i++) {
  // assert: p does not start at positions 0..i-1 of s
  for (j=0; j<p.length(); j++) {
    // assert: s[i..i+j-1] matches first j chars of p
    ...   // break if match fails somewhere
}}
```

? How do we determine the result?

Naive String Search

Precondition: s is a nonempty string, and p is a nonempty pattern string.
Postcondition: true is returned iff p is found in s.

```
bool found = false;
for (i=0; i<s.length()-p.length() && !found; i++) {
  // assert: p does not start at positions 0..i-1 of s
  for (j=0; j<p.length(); j++) {
    // assert s[i..i+j-1] matches first j characters of
    ...  // break if match fails somewhere
    }
  if (j==p.length()) found=true;
return found;
```

Naive String Search

Precondition: s is a nonempty string, and p is a nonempty pattern string.
Postcondition: true is returned iff p is found in s.

```
bool found = false;
for (i=0; i<s.length()-p.length() && !found; i++) {
  // inv: p does not start at positions 0..i-1 of s
  for (j=0; j<p.length(); j++) {
    // inv: s[i..i+j-1] matches first j characters of p
    if (s[i+j] != p[j]) break;  // exit inner loop
    }
  if (j==p.length()) found=true;
return found;
```

- An assertion at the start of loop body is called a loop **invariant**.
- Not good SW engineering to use break (see exercises)
- Many much more efficient string search algorithms exist

Exercises

- Write a function, `string rotate(sting s)` that rotates it right by 1 character.
- Modify the above to rotate a string by `num` characters.
- Modify the above to rotate s itself instead of returning a string.
- Modify the string search algorithm to avoid the break (without using extra returns). Hint: use a boolean variable.
- Modify the string search algorithm to return the position of the match instead of a boolean. Also change the precondition to allow empty s and p.
- Write a function, `bool anagram(string s1, string s2)` that returns true or false depending on whether or not s1 and s2 are anagrams of each other.
- Write a function `void removeDups(string s)` that removes duplicated characters in a string. For example, "to beeeee or nnot to be" would be changed to "to be or not to be".