# Strings

July 29, 2021

# 1 C++ Strings

## 1.1 Topics

- string library
- stiring objects and methods
- string operators
- slicing string
- string traversal
- comparing and updating strings

## 1.2 String review

- we've used string library to declare string variables in earlier chapters
- we've seen few examples of string applications over the chapters
- quick revieew

### 1.2.1 C-string

- uses array concept; which we don't know yet!
- C array is not easy to work with though important concept to master

[1]:

```
#include <iostream>

using namespace std;
```

[3]:

```
// C way to declare string - painful to work with!
// array of characters
char text[] = "this is a c-string";
```

[3]:

```
cout << "text = " << text << endl;
```

```
text = this is a c-string
```

- `cin` and other operations on c-strings are not easier without knowing array and pointers well

### 1.2.2 C++ string objects

- std::string is a std::basic_string<char> template type defined in **string** header
- more: https://en.cppreference.com/w/cpp/string/basic_string

- string is an advanced type of container with many member variables and member functions
  - variables of advanced type are called objects
  - advanced types' provide a number of member functions are called methods
  - one can define any user-defined type using **struct** or **class** that we'll learn in later chapters

[1]:

```
// C++ string
#include <string>

using namespace std;

// declare a string variable/object
string first;
```

[2]:

```
// assigning string value to string object
first = "Hello, ";
```

[3]:

```
// declare and initialize string object
string second = "World";
```

### 1.3 Inputting and Outputting strings

- printing string objects and literal to standard output/console/monitor
- use `<iostream>` and `std namespace`
- syntax:

```
cout << strObject << "string literal << ...;
```

- inputting string data from from standard input/keyboard
- syntax:

```
cin >> strVar >> strVar2 >> ...; // read individual word
getline(cin, strVar); // read a sinle line with spaces
```

[4]:

```
// output string literals and objects
#include <iostream>
using namespace std;

cout << first << second << "!" << endl;
```

```
Hello, World!
```

[5]:

```
cout << "Enter your first name and last name: ";
cin >> first >> second;
cout << "Hello, " << first << " " << second << "!";
```

```
Enter your first name and last name: John Smith
Hello, John Smith!
```

[7]:

```
cout << "Enter your full name: ";
getline(cin, first);
cout << "Hello, " << first << "!" << endl;
```

```
Enter your full name: John C Smith
Hello, John C Smith!
```

## 1.4 C++ strings and variables

- this chapter goes more in depth on string data
- string variable is a container for a sequence of 0 or more characters
  - characters are anything from set of:
    - symbols (%, &, $, etc.)
    - alphabets (a, B, x, etc.)
    - digits (1, 9, 0, etc.)
- in C++, string is represented using a pair of double quotes ("")

- string is made up of ordered sequence of character elements as depicted in the following figure
- each character has an internal indexing or placing we can refer to it by its index

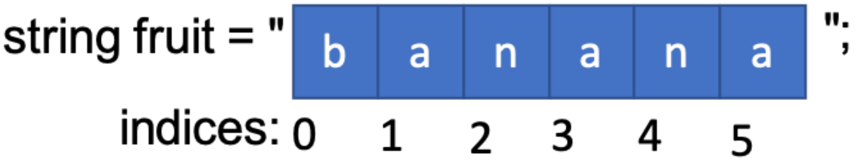


Fig. C++ string representation

### 1.4.1 C-string example

- no need to include any library to use C-string

## 1.5 Object Members

- there are some member variables and many membmer functions (called methods) available in string objects
- a complete list is provided in this reference: https://en.cppreference.com/w/cpp/string/basic_string
- we'll go over some commonly used methods with examples in this notebook
- syntax to access members of objects:

```
object.member_variable
object.member_function()
```

- we use . (dot) member access operator to access object's members

## 1.6 Element access

- extracting and updating characters

- the following member functions/methods let's you access element:
  - **at(index)** - access the specified character at index with bounds checking
  - **operator[index]** - access the specified character at index without bounds checking
  - **front( )** - access the first character
  - **back( )** - access the last character
- index must be a valid index between **0 to length-1**

[35]:

```
string fruit = "banana";
```

[36]:

```
char first_letter;
```

[37]:

```
// access the first character at index 0
first_letter = fruit.at(0);
```

[38]:

```
cout << "first letter of " << fruit << " is " << first_letter << " = " <<␣
 ↪fruit[0];
```

```
first letter of banana is b = b
```

[39]:

```
//second character
cout << "second character = " << fruit[1] << " = " << fruit.at(1);
```

```
second character = a = a
```

[ ]:

```
// there are 6 characters in banana
cout << "last character = " << fruit[6];
// [] - doesn't check the bound; output is undetermined
```

[41]:

```
// at() - checks the bounds; throws runtime-error
cout << "last character = " << fruit.at(6);
```

```
last character =


        Standard Exception: basic_string
```

[42]:

```
cout << "front = " << fruit.front() << " and back = " << fruit.back();
```

```
front = b and back = a
```

### 1.6.1 Updating string in place

- string is a mutable type that can be changed in place!
- using [ ] - member access operator, we can assign new character at some index
  - index must be a valid one between [**0 ... length-1**]

[43]:

```
// capitalize the first character by replacing 'b' with 'B'
fruit[0] = 'B';
```

[44]:

```cpp
cout << "I love, " << fruit << "!";
```

```
I love, Banana!
```

## 1.7 Capacity

- knowing the length of a string (numbers of characters) helps with many operations
- the following methods provide capacity of string objects:
  - **length()** or **size()** - returns the number of characters
  - **empty()** - checks whether the string is empty

[45]:

```cpp
cout << "length of " << fruit << " = " << fruit.size() << " = " << fruit.
 ↪length();
```

```
length of Banana = 6 = 6
```

[48]:

```cpp
cout << "is fruit empty? " << boolalpha << fruit.empty();
```

```
is fruit empty? false
```

## 1.8 Traversal

- traversing a string is a common task where you access every character from first to the last
- there are several ways to traverse a string

[96]:

```cpp
// using capacity to traverse/iterate over a string
for(int i=0; i<fruit.length(); i++) {
    cout << "fruit[" << i << "] = " << fruit[i] << endl;
}
```

```
fruit[0] = B
fruit[1] = a
fruit[2] = n
fruit[3] = a
fruit[4] = n
fruit[5] = a
```

[95]:

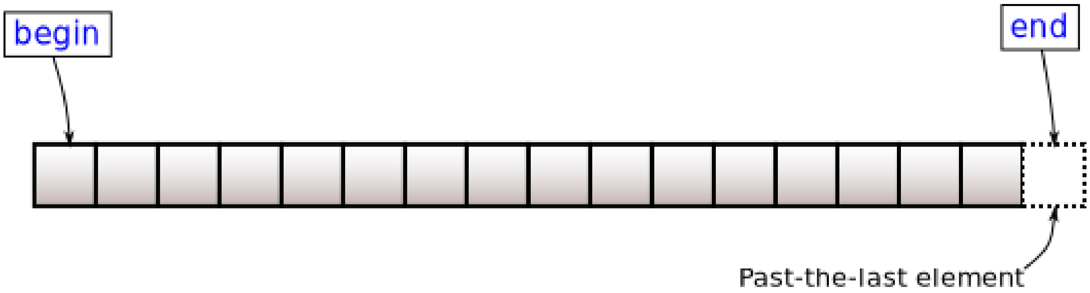
```cpp
#include <cctype>

for(auto ch: fruit)
    cout << ch << " -> " << char(toupper(ch)) << endl;
```

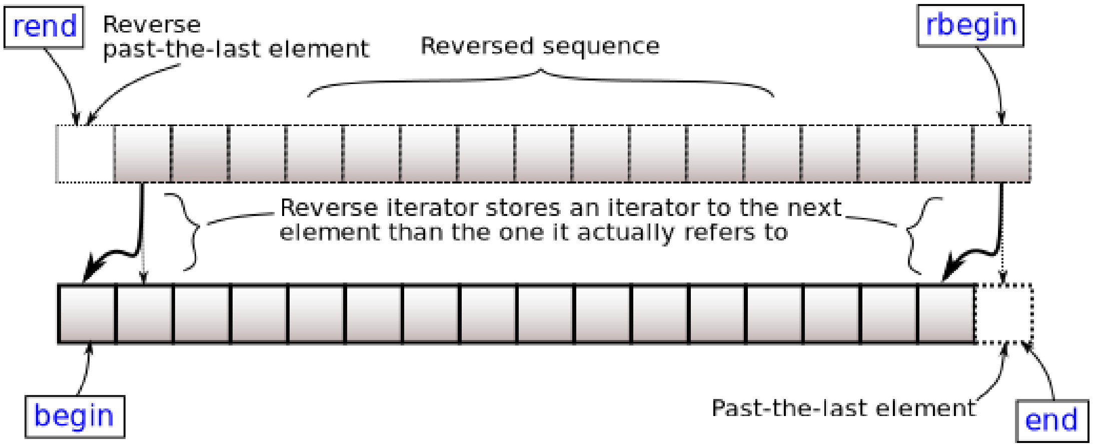
```
B -> B
a -> A
n -> N
a -> A
n -> N
a -> A
```

### 1.9 Iterators

- iterators are special pointers that let you iterate over or traverse a string
- the following methods return an iterator:
    - **begin( )** - returns a forward iterator to the beginning
    - **end( )** - returns a forward iterator to the end
    - **rbegin( )** - returns a reverse iterator to the beginning
    - **rend( )** - returns a reverse iterator to the end
- the following figure demonstrates begin() and end() iterators



- the following figure demonstrates rbegin() and rend() iterators


rend
Reverse
past-the-last element
Reversed sequence
rbegin
Reverse iterator stores an iterator to the next
element than the one it actually refers to
begin
Past-the-last element
end

[20]:
```
// automatically determine the type of iter which is a forward iterator
auto iter = fruit.begin();
```

[21]:
```
// what is iter pointing to?
cout << *iter;
```

```
B
```

[22]:
```
// increment iterator by one element
iter += 1;
```

[23]:
```
cout << *iter;
```

```
a
```

[24]:

```cpp
// forward iterator
for(auto it=fruit.begin(); it != fruit.end(); it += 1) {
    cout << *it << " ";
}
```

```
B a n a n a
```

[25]:

```cpp
// reverse iterator
for(auto it=fruit.rbegin(); it!=fruit.rend(); it++) {
    cout << *it << " ";
}
```

```
a n a n a B
```

## 1.10 Operations

- string objects have a bunch of methods to perform various common operations on string data
- the following are some commonly used operations:

### 1.10.1 clear

- clears the contents making string object empty!

[4]:

```cpp
string strData = "Pirates of the Carribean!";
```

[5]:

```cpp
// clear the content
strData.clear();
cout << "strData = " << strData;
```

```
strData =
```

### 1.10.2 insert

- insert a character or string at some given index
- **insert(index, count, char)** insert `count` characters at some `index`
- **insert(index, string)** - insert some `string` at `index`

[5]:

```cpp
strData = "Pirates of the Carribean!";
```

[7]:

```cpp
// insert 1 $ at index 0
strData.insert(0, 1, '$');
```

[8]:

```cpp
cout << "strData = " << strData;
```

```
strData = $Pirates of the Carribean!
```

[10]:

```
// insert 5 astersisks at index 5
strData.insert(5, 5, '*');
```

[11]:

```
cout << "strData = " << strData;
```

```
strData = $Pira*****tes of the Carribean!
```

[12]:

```
strData.insert(0, "The ");
```

[13]:

```
cout << "strData = " << strData;
```

```
strData = The $Pira*****tes of the Carribean!
```

### 1.10.3 erase

**erase(index, count)** - erases `count` characters starting from `index`

[14]:

```
// erase all 5 asterisks starting at index 9
strData.erase(9, 5);
```

[15]:

```
strData
```

[15]:

```
"The $Pirates of the Carribean!"
```

### 1.10.4 append

- the following methods append characters to the end of string objects
  - **push_back(ch)** - appends a character to the end
  - **append(str)** - appends string to the end
  - **operator+=** - appends string to the end

[3]:

```
string some_str;
```

[4]:

```
some_str = "";
```

[5]:

```
some_str.push_back('1');
some_str.append("2");
some_str += "3456";
```

[6]:

```
some_str
```

[6]:

```
"123456"
```

## 1.11 Search

- searching for a substring is often a common task performed with strings data
- also refered to as “finding a needle in the haystack”
- `find` and `rfind` methods help in finding a substring in some string

### 1.11.1 find( str, [pos] )

- finds the first `str` in the string starting from `pos`
  - if no `pos` is provided, first index, 0 is used
- if `str` is found, returns begining position/index of `str`
- if str is not found, returns `npos` constant defined in `string::` namespace
  - `npos` is the largest possible value for **size_t**; system dependent

[7]:

```
// what is npos?
cout << string::npos;
```

```
18446744073709551615
```

[8]:

```
string haystack, search_str;
size_t found;
```

[9]:

```
haystack = "There are maanny needles or just a few needle in the haystack!";
```

[10]:

```
search_str = "needle"; // TODO: change this to "Needle" and find
```

[10]:

```
"needle"
```

[11]:

```
found = haystack.find(search_str);
```

[12]:

```
cout << found;
```

```
17
```

[13]:

```
// check if substring is found or not
if (found == string::npos)
    cout << search_str << " NOT found!\n";
else
    cout << search_str << " found at: " << found << endl;
```

```
needle found at: 17
```

### 1.11.2 rfind( str, [pos] )

- search the first substring in backward direction starting from `pos`
  - if no `pos` is provided, last index is used

[14]:

```
found = haystack.rfind(search_str);
// check if substring is found or not
if (found == string::npos)
    cout << search_str << " NOT found!\n";
else
    cout << search_str << " found at: " << found << endl;
```

```
needle found at: 39
```

### 1.11.3 replace

- replaces the part of string indicated by `index` with a new string
- **replace(index, count, newStr)**
  - replace some string from `index` to `index+count` by `newStr`

```
[15]: some_str = "12345abc";
```

```
[16]: some_str.replace(0, 1, "A");
```

```
[17]: some_str
```

```
[17]: "A2345abc"
```

```
[18]: some_str.replace(1, 5, "B");
```

```
[19]: some_str
```

```
[19]: "ABbc"
```

```
[20]: // insert with replacing 0 character
      some_str.replace(1, 0, "WXYZ");
```

```
[21]: some_str
```

```
[21]: "AWXYZBbc"
```

### 1.11.4 Search and replace application

- a commmon feature provided by text editors

```
[23]: // let's see the contents of haystack
      haystack
```

```
[23]: "There are maanny needles or just a few needle in the haystack!"
```

```
[28]: // let's search misspelled word "maanny" and replace with "many"
      size_t wordIndex = haystack.find("maanny")
```

```
[29]: wordIndex
```

```
[29]: 10
```

```
[31]: haystack.replace(wordIndex, string("maanny").length(), "many")
```

```
[31]: "There are many needles or just a few needle in the haystack!"
```

[32]:
```
// replace the first needle with poodle
haystack.replace(haystack.find("needle"), 6, "poodle")
```

[32]:
```
"There are many poodles or just a few needle in the haystack!"
```

## 1.12 Sub string

- **substr(pos, count)** returns a substring from `pos` index to `pos+count` index
  - if count is not provided, returns to the end or **npos**
  - `npos` is a constant value defined in `string::` namespace

[64]:
```
// return from index 1 to the end or npos
cout << some_str.substr(1);
```

```
WXYZB
```

[74]:
```
// return 4 characters starting from 1
cout << some_str.substr(1, 4);
```

```
WXYZ
```

## 1.13 String comparisons

- two string values can be compared using comparison operators
- all comparison operators (==, !=, <, <=, >, >=) are overloaded to work with string types
- strings are compared character by character using ASCII value

[97]:
```
string a = "apple";
```

[98]:
```
string b = "ball";
```

[104]:
```
string c = "Apple";
```

[100]:
```
// both size and values must be equal!
if (a == b) // every character in var 'a' must equal to corresponding character in var 'b'
    cout << a << " equals to " << b << endl;
else
    cout << a << " is NOT equal to " << b << endl;
```

```
apple is NOT equal to ball
```

[102]:
```
if (a <= b)
    cout << a << " comes before " << b << endl;
else
    cout << a << " doesn't come before " << b << endl;
```

```
apple comes before ball
```

[106]:

```
if (a <= c)
    cout << a << " comes before " << c << endl;
else
    cout << a << " doesn't come before " << c << endl;
```

```
apple doesn't come before Apple
```

## 1.14 Numeric conversions

- strings can be converted into numeric values (integers or floating points) as appropriate

### 1.14.1 string to signed integers

- **stoi( ), stol( ), stoll( )** - converts a string to a signed integers

[107]:

```
cout << stoi("123");
```

```
123
```

[117]:

```
cout << stoi("-454532") << " " << stol("-45352343441 asdf") << " " <<␣
 ↪stoll("552353253 adsfasf");
```

```
-454532 -45352343441 552353253
```

### 1.14.2 string to unsigned integers

**stoul( ), stoull( )** - converts a string to unsigned integer

[118]:

```
cout << stoul("454532") << " " << stoull("-45352343441 text");
```

```
454532 18446744028357208175
```

### 1.14.3 string to floaing point value

- **stof( ), stod( ), stold( )** - converts a string to floating point value

[119]:

```
cout << stof("-454532") << " " << stof("-453.123 text") << " " << stof("552.34␣
 ↪adsfasf");
```

```
-454532 -453.123 552.34
```

[120]:

```
// throws run-time error
cout << stof("a5235");
```

```
        Standard Exception: stof: no conversion
```

[6]:

```
cout << stod("-454532") << " " << stod("-453.123 text") << " " << stod("552.34␣
 ↪adsfasf");
```

```
-454532 -453.123 552.34
```

### 1.14.4 integral or floating point value to string

- **to_string( )** converts integral or floats to string

[123]:
```
string new_str = to_string(123).append("456");
```

[124]:
```
new_str
```

[124]:
```
"123456"
```

[5]:
```
cout << (to_string(345.44545)).append(" some text");
```

```
345.445450 some text
```

## 1.15 Dynamic string variables

- pointers can point to string types
- string pointers can be used to allocate dynamic memory in heap

[1]:
```
#include <iostream>
#include <string>

using namespace std;
```

[2]:
```
string full_name = "John Doe";
string * ptr_full_name = &full_name;
```

[3]:
```
// dereference ptr_full_name
cout << full_name << " == " << *ptr_full_name;
```

```
John Doe == John Doe
```

[4]:
```
// allocate dynamic memory in heap and initialize it with data
string * ptr_var = new string("Jake Smith");
```

[5]:
```
cout << *ptr_var;
```

```
Jake Smith
```

[6]:
```
// assign new value to *ptr_var
*ptr_var = "Jane Fisher";
```

### 1.15.1 String Application - Convert Decimal into Binary

- Define a function that takes an integer and returns the binary representation of the integer.
  - e.g. $10_{10} = 1010_2$
- let's use algorithm defined in Chapter 02 and the partial code in Chapter 03:

1. repeteadly divide the decimal number by base 2 until the quotient becomes 0
2. collect the remainders in reverse order
    - the first remainder becomes the last bit (least significant) in binary

[1]:
```
#include <iostream>
#include <string>

using namespace std;
```

[2]:
```
string binary(unsigned int decimal) {
    // decimal to binary conversion requires to calculate both quotient and␣
 ↪remainder
    const int divisor = 2; // divisor is contant name whose value can't be␣
 ↪changed once initialized with
    int dividend;
    int quotient, remain;
    string answer = ""; // collect remainders by prepending as a string
    quotient = decimal;

    while(quotient != 0) { // we can programatically check when the loop should␣
 ↪exit
        // repeated computation
        dividend = quotient;
        remain = dividend%divisor;
        quotient = dividend/divisor;
        // print intermediate results; help us see and plan further computation
        //cout << dividend << '/' << divisor << " => quotient: " << quotient <<␣
 ↪" remainder: " << remain << endl;
        answer = to_string(remain) + answer; // prepend remainder to answer
    }
    if (answer == "")
        return "0";
    return answer;
}
```

[4]:
```
cout << "10 decimal in binary = " << binary(10) << endl;
```

```
10 decimal in binary = 1010
```

### 1.15.2 Convert Binary into Decimal

- algorithm steps as provided in Data, Variable and Operations chapter:
    1. multiply each binary digit by its place value in binary
    2. sum all the products
- Define a function that takes a binary number provided in string and converts into decimal representation
    - E.g. $1010_2 = 10_{10}$

[7]:

```cpp
#include <cmath>
#include <iostream>
#include <string>

using namespace std;
```

[8]:

```cpp
unsigned int decimal(string binary) {
    int answer = 0;
    int digitCount = binary.size();
    for(int i=0; i<digitCount; i++) {
        if (binary[i] == '0') continue;
        int placeValue = digitCount-i-1;
        answer += pow(2.0, placeValue);
    }
    return answer;
}
```

[9]:

```cpp
cout << "1010 in binary = " << decimal("1010") << " in decimal." << endl;
```

```
1010 in binary = 10 in decimal.
```

## 1.16 Labs

1. Read and solve the Kattis problem Hissing Microphone - https://open.kattis.com/problems/hissingmicrophone
   - use partial solution file `hissing.cpp` in labs/hissingmicrophone folder
   - use Makefile provided to compile the file
   - fix all the FIXMEs and write #FIXED next to each FIXME once fixed

## 1.17 Exercises

1. Write a function that checks if a given string has at least one digit (0-9) in it.
   - Write 3 automated test cases

[8]:

```cpp
// Exercise 1 Sample Solution
#include <iostream>
#include <string>
#include <cstring>
#include <cassert>

using namespace std;
```

[9]:

```cpp
bool hasDigit(string text) {
    for(char ch: text) {
        if (isdigit(ch)) return true;
    }
    return false;
}
```

[10]:

```
// test hasDigit
void test_hasDigit() {
    assert(hasDigit("some text with d1g1t!") == true);
    assert(hasDigit("this text has no digit") == false);
    assert(hasDigit("24242") == true);
    cerr << "all test cases passed for hasDigit()\n";
}
```

[11]:

```
test_hasDigit();
```

```
all test cases passed for hasDigit()
```

2. Convert Exercise 1 into a complete program
    - prompt user to enter some text
    - make program continue to run until the user wants to quit
3. Write a function that checks if a given string is a palindrome. Palindromes are words and phrases that read the same backward as forward such as **madam, race car, etc.**
    - more on Palindromes: https://en.wikipedia.org/wiki/Palindrome
    - it's okay if the function works for word only
    - challenge yourself to make it work for phrases as well
    - ignore cases (i.e., A equals a)
    - write at least 3 automated test cases

[1]:

```
// Sample solution for exercise #3
#include <iostream>
#include <string>
#include <cstring>
#include <cassert>

using namespace std;
```

[2]:

```
/*
palindromic texts: A, AA, ABA, ABBA

Algorithm steps:
1. for each character up to the middle one in a given phrase
    ii. compare the corresponding characters from left and right of the phrase
        a. do a case insensitve comparision
    iii. if a single corresponding pair is not equal, the phrase is NOT␣
 ↪palindrome
    iv.  if all the pairs match, the word is palindrome
*/

bool isPalindrome(string word) {
    int left_index = 0; // index from the beginning of the word
    int right_index = word.length()-1; // index from the end of the word
    int mid = word.length()/2; // mid index to stop the comparison
    bool mismatched = false;
    while(left_index < mid && !mismatched) { // stop before the mid index or any pair mismatched
        // convert to lowercase to make case insensitive comparison
        char left_char = tolower(word[left_index]);
        char right_char = tolower(word[right_index]);
        // if no match, set the mismatched flag to true;
        if (left_char != right_char) mismatched = true;
        // if they match, move the indices to point the next pair
        left_index++;
        right_index--;
    }
    // if mismatched return false; else all pairs must have matched, return true
    return mismatched? false : true;
}
```

[3]:

```
void test_isPalindrome() {
    assert(isPalindrome("") == true); // empty string is a plindrome!?!?
    assert(isPalindrome("A") == true);
    assert(isPalindrome("AB") == false);
    assert(isPalindrome("ABA") == true);
    assert(isPalindrome("ABBA") == true);
    assert(isPalindrome("racecar") == true);
    assert(isPalindrome("race car") == false);
    cerr << "all test cases passed for isPalindrome()\n";
}
```

4. Convert Exercise 3 into a complete program.
    - program prompts user to enter a string
    - determines and lets the user know if the string is a palindrome or not
    - program continues to run until the user wants to quit
5. Improve Exercise 4 to ignore punctuations including spaces!
    - if you named the improved isPalindrome function as isPalaindromeV1,
        - the following test cases must pass!

[ ]:

```
/*
palindromic texts: A, AA, ABA, ABBA, "race car"

Algorithm steps:
1. for each character up to the middle one in a given phrase
    i. ignore all the non-alphabetic characters on both ends of the phrase
    ii. compare the corresponding characters from left and right of the phrase
    iii. if a single pair is not equal, the phrase is NOT reversible
    iv.  if all the pairs match, the word is reversible
*/
bool isPalindromeV1() {
    // FIXME using the above algorithm
    return true;
}
```

[ ]:

```
void test_isPalindromeV1() {
    assert(isPalindromeV1("") == true); // empty string is a plindrome!?!?
    assert(isPalindromeV1("A") == true);
    assert(isPalindromeV1("AB") == false);
    assert(isPalindromeV1("ABA") == true);
    assert(isPalindromeV1("ABBA") == true);
    assert(isPalindromeV1("racecar") == true);
    assert(isPalindromeV1("race car") == true); // ignore white spaces...
    cerr << "all test cases passed for isPalindromeV1()\n";
}
```

6. Write a program that counts the number of vowels (a, e, i, o, u) and consonants (alphabets except vowels) in a given text.
    - program promps user to enter the text
    - program should account for both upper and lower case alphabets
    - program should continue to run until the user wants to quit
7. Write a program that checks the strength of the given password.
    - use a scoring system based on the varieties of character type present as described below:
    - 1 point if it contains at least 1 lowercase
    - 1 point if it contains at least 1 uppercase
    - 1 point if it contains at least 1 digit
    - 1 point if it contains at least 1 symbol from the group (~!@#$%^&*()_-+={})
    - 1 point if the length of the password is 8 characters or long
    - interpretation of total points (max 5):
    - if points is 5 or more - Excellent
    - if points is 3 or more - Good
    - if points is 2 or less - Bad

### 1.18 Kattis problems

- there are a lot of Kattis problems on text/string manipulation
- some simple problems are listed below
- solve each problem using function(s) so that you can write at least 3 test cases for function(s) used as part of the solution

1. Hissing Microphone - https://open.kattis.com/problems/hissingmicrophone

2. Avion - https://open.kattis.com/problems/avion

3. Apaxiaaaaans! - https://open.kattis.com/problems/apaxiaaans

4. Alphabet Spam - https://open.kattis.com/problems/alphabetspam

5. Simon Says - https://open.kattis.com/problems/simonsays

6. Simon Says - https://open.kattis.com/problems/simon

7. Fifty Shades of Pint - https://open.kattis.com/problems/fiftyshades

8. Quick Brown Fox - https://open.kattis.com/problems/quickbrownfox

9. Encoded Message - https://open.kattis.com/problems/encodedmessage
10. Trik - https://open.kattis.com/problems/trik
11. Digit Product - https://open.kattis.com/problems/sifferprodukt
12. Magic Trick - https://open.kattis.com/problems/magictrick
13. FYI - https://open.kattis.com/problems/fyi
14. Methodic Multiplication - https://open.kattis.com/problems/methodicmultiplication
    - simple multiplication

## 1.19 Summary

- this chapter covered C++ string type
- delcare and use string type
- various operations and member functions or methods provided to string objects
- exercises and sample solutions

[ ]: