Ch09-Strings

August 8, 2020

1 9 Strings

1.1 Topics

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

1.2 9.1 string and variables

- we’ve used string library to declare string variables in earlier chapters
- we’ve seen few examples of string applications over the chapters
- this chapter covers goes in depth about string data
- string variable is a container for a sequence of 0 or more characters
  - characters are anything from 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 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

1.2.1 c-string variables

[2]:
```
#include <iostream>

using namespace std;
```

[3]:
```
// C way to declare string - painful to work with!
// array of characters; we don't know array yet!!
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

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 members variable and member functions
  - variables of advanced type are called objects
  - member functions are called method
  - one can define any type using **struct** or **class** that we'll learn later

[4]:

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

using namespace std;

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

[5]:

```
// assing string value to string variable
first = "Hello, ";
```

[6]:

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

[7]:

```
// out put string literals and variables
#include <iostream>

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

```
Hello, World!
```

1.3 9.2 Member functions

- there are many membmer functions and 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 ones with examples
- syntax to access members from objects:

```
object.data_member
object.member_function()
```

- we use . (dot) member access operator

1.4 9.3 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
```

1.4.1 updating string in place

- string is mutable type; that can be changed in place!
- using [ ] operator, we can assign new character at some index
  - index must be a valie one [**0 … length-1**]

[43]:
```
// capitalized the first character by replacing b with B
fruit[0] = 'B';
```

[44]:
```
cout << "I love, " << fruit << "!";
```

```
I love, Banana!
```

1.5 9.4 Capacity

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

[45]:

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

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

[48]:

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

```
is fruit empty? false
```

1.6 9.5 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]:

```
// 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]:

```
#include <cctype>

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

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

1.7 9.6 Iterators

- iterators are special pointers that let you iterate or traverse a string

- the following methods retrun 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

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

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

[25]:
```
// reverse iterator
for(auto it=fruit.rbegin(); it!=fruit.rend(); it++) {
    cout << *it << " ";
}
```

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

1.8 9.7 Operations

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

1.8.1 clear

- clears the contents; making string object empty!

[3]:
```
string strData = "Pirates of the Carribean!";
```

[4]:
```
// clear the content
strData.clear();
cout << " strData = " << strData;
```

```
 strData =
```

1.8.2 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]:
```
strData = "Pirates of the Carribean!";
```

[7]:
```
// insert 1 $ at index 0
strData.insert(0, 1, '$');
```

[8]:
```
cout << "strData = " << strData;
```

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

[10]:
```
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!
```

1.8.3 erase

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

[14]:
```
// erase all 5 asterics
strData.erase(9, 5);
```

[15]:
```
strData
```

[15]:
```
"The $Pirates of the Carribean!"
```

1.8.4 append

- the following methods append characters to the end
  - **push_back(ch)** - appends a character to the end
  - **append(str)** - appends string to the end

    - **operator+=** - appends string to the end

[ ]:
```
string some_str;
```

[55]:
```
some_str = "";
```

[56]:
```
some_str.push_back('1');
some_str.append("2");
some_str += "3456";
```

[57]:
```
some_str
```

[57]:
```
"123456"
```

1.8.5 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

[58]:
```
some_str.replace(0, 1, "A");
```

[59]:
```
some_str
```

[59]:
```
"A23456"
```

[60]:
```
some_str.replace(1, 5, "B");
```

[61]:
```
some_str
```

[61]:
```
"AB"
```

[62]:
```
// insert with replacing 0 character
some_str.replace(1, 0, "WXYZ");
```

[63]:
```
some_str
```

[63]:
```
"AWXYZB"
```

1.9 9.8 sub string

- **substr(pos, count)** returns a substring from pos index to pos+count index
    - if count is not provides, returns to the end or **npos**
    - **npos** is a constant value that's the largest possible index for string objects
        - largest possible value for **size_t**

[73]:
```
// what is npos?
cout << string::npos;
```

```
18446744073709551615
```

[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
```

1.10 9.9 Search

- searching for a substring is often a common task performed with strings data
- also refered to as finding needle in haystack
- following methods help in finding substrings in strings:

1.10.1 find(str, pos)

- finds the first substring in the string starting from pos
  - if no pos is provided, first index is used
- returns position of the first character of the found substring or **npos** if no such substring is found

[75]:

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

[79]:

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

[87]:

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

[88]:

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

[89]:

```
cout << found;
```

```
15
```

[90]:

```
// 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: 15
```

1.10.2 rfind(str, pos)

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

[91]:

```
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: 37
```

1.11 9.10 string comparisons

- two string values can be compared using comparison operators
- operators (==, !=, <, <=, >, >=) are all 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 a must equal to corresponding character in 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
```

1.12 9.11 Numeric conversions

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

1.12.1 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
```

1.12.2 string to unsigned integers

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

[118]:

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

```
454532 18446744028357208175
```

1.12.3 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
```

1.12.4 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
```

1.13 9.12 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";
```

1.14 9.13 Exercises

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

[8]:

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

using namespace std;
```

[9]:

```
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 string
    - 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; but not phrases
    - 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 pair is not equal, the phrase is NOT reversible
    iv.  if all the pairs match, the word is reversible
*/
```

```
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]:

```
test_isPalindrome();
```

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

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

1.15 9.14 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 each function 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

1.16 9.15 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

[ ]: