CS224 Object Oriented Programming and Design Methodologies Lab Manual

Lab 3- Strings and Pointers

Department of Computer Science

Dhanani School of Science and Engineering

Habib University

Fall 2025

Contents

Lab 3

Strings and Pointers

3.1 Guidelines

1. Use of AI is strictly prohibited. This is not limited to the use of AI tools for code generation, debugging, or any form of assistance. If detected, it will result in immediate failure of the lab, student will be awarded 0 marks, reported to the academic integrity board and appropriate disciplinary action will be taken.
2. Absence in lab regardless of the submission status will result in 0 marks.
3. All assignments and lab work must be submitted by the specified deadline on Canvas. Late submissions will not be accepted.

3.2 Objectives

The objectives of this lab are to help students gain practical experience with strings, arrays, and pointers in C++, and to apply them in computational problems. Specifically, students will learn to:

1. Work with C-style strings
   - Store and initialize strings using character arrays.
   - Access string characters using array subscripting and pointers.
   - Read strings from the user using `cin` and `getline()`.
   - Display strings using pointers and understand the role of the null character.
2. Manipulate pointers
   - Declare and initialize pointer variables.
   - Use the address-of (`&`) and dereference (`*`) operators.
   - Traverse arrays using pointer arithmetic instead of array indexing.
   - Understand null pointers and safe pointer usage.
3. Work with arrays
   - Store and process arrays of characters and integers.
   - Sort arrays of strings and numerical values using pointers.
   - Perform computations such as minimum, maximum, average, standard deviation, mean, median, and mode.
   - Implement functions that return multiple values via pointers and indicate presence with boolean flags.

3.3 Directory Structure

Labs will have following directory structure:

```
lab
├── manual
│   └── lab.pdf
├── src
│   ├── *.hpp
│   └── *.cpp
├── tests (optional)
│   └── *.cpp
└── makefile (optional)
```

`manual` will contain the lab manual pdf. `src` will contain the source code files, and `tests` (if present) will contain the test files. `makefile` (if present) will contain the makefile for testing and running.

3.4 Strings

You have previously been introduced to single characters in `C++`. We can now continue with strings. A string is a series of characters stored in consecutive memory locations. `C++` supports two ways to handle strings. The first is the same as in C, that is, with an array of characters (C-style string). A C-style string must end with a special character, the null character, which identifies the end of the string. It is the first character in the ASCII set, its ASCII value is 0 and is denoted by `'\0'`.

3.4.1 String Literals

A string literal is a sequence of characters enclosed in double quotes. A string literal is a constant. In particular, C++ treats it as a nameless constant character array. The quotes are not considered part of the string, but serve only to delimit it. In particular, when the compiler encounters a string literal, it allocates memory to store its characters plus the null character to mark its end. For example, if the compiler encounters the string literal "text", it allocates five bytes to store the four characters of the string plus one for the null character. A string literal may be empty. For example, the string literal `""` is stored as a single null character.

```cpp
#include <iostream>

int main() {
    const char *s1 = "text"; // stored as {'t','e','x','t','\0'}
    const char *s2 = "";     // stored as {'\0'} (empty string)

    std::cout << s1 << std::endl; // prints: text
    std::cout << s2 << std::endl; // prints nothing (empty)

    return 0;
}
```

Listing 3.1: String literals in C++

3.4.2 Storing Strings

To store a C-style string in a variable, we use an array of characters. Because a C-style string ends with the null character, to store a string of N characters the size of the array should be N+1 at least. For example, to store a string of up to 4 characters, we write:

```cpp
char str[5];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```cpp
char str[5] = "text";
```

In fact, this declaration is equivalent to:

```
char str[5] = {'t', 'e', 'x', 't', '\0'};
```

because the ASCII value of `'\0'` is 0, `str[0]` becomes `'t'`, `str[1]` becomes `'e'` and the rest elements are initialized to 0, or equivalently to `'\0'`. Similarly, with the declaration: As with an ordinary array, if the number of the characters is less than the size of the array, the remaining elements are initialized to 0. If it is greater, the compilation fails. For example, with the declaration:

```
char str[5] = "te";
```

A flexible way to initialize the array is to omit its length and let the compiler compute it. For example, with the declaration:

```
char str[] = "text";
```

The compiler calculates the length of "text" and then allocates five bytes for str to store the four characters plus the null character. If we use sizeof(str) to output its length, we'll see that it is 5, indeed. Leaving the compiler to compute the length, it is easier and safer, since counting by hand can lead to a calculation error or we may forget to add an extra place for the null character.

3.4.3 Writing Strings

There are many ways to display a string. For example, we can use cout and pass a pointer to the string. The following code uses the name of the array as a pointer to the first character of the string:

```
1 char str[] = "Print a message";
2 std::cout << str;
```

The code displays the characters of the string beginning from the character that the pointer points to until it encounters the null character. If we don't want to display the string from the beginning we make the pointer point to the desired position. For example, if we write: `cout << str+8;` or equivalently: `cout << &str[8];` it displays the part beginning from the ninth character, that is, *message*. If the null character is encountered, no more characters are printed. For example, if we add the statement: `str[5] = '\0';` before the output, the code displays *Print*.

If cout does not encounter the null character, it continues past the end of the string until it finds a null character somewhere next in the memory. For example:

```
#include <iostream>

int main() {
    char str[10];
    str[0] = 'a';
    str[1] = 'b';
    std::cout << str << std::endl;

    return 0;
}
```

The program displays 'a' and 'b' and then garbage characters until it finds the null character or an illegal memory access occurs. If we write `char str[10] = 0;` then, because the elements are initialized to 0, the program would display *ab*.

3.4.4 Reading Strings

There are many ways to read a string. For example, we can use `cin` to read it:

```
#include <iostream>

int main() {
    char str[10];
    int i = 5;
    std::cout << "Enter text: ";
    std::cin >> str; // Danger, be careful.
    std::cout << str << " " << i << std::endl;

    return 0;
}
```

The program reads a string, stores it into the str array and displays it. By default, `cin` reads characters until it encounters a white space character (i.e., space, tab, or new line character). Then, it appends a null character at the end of the string.

Note that if the user enters many words (e.g., nice day), because `cin` stops reading characters when it encounters a white character such as the space, only the first word will be stored in the array (e.g., nice). The remaining characters are left in the input queue. To be able to read strings that contain spaces we can use the `getline()` function, which is a member of the `istream` class.

```
    char str[30];
    cin.getline(str, sizeof(str));
```

`getline()` will read a maximum of 29 characters, leaving one place for the null character. The characters will be stored in the str array. If the new line character is encountered earlier, `getline()` stops reading. Suppose that the user enters an integer and then presses Enter. What does the following program output?

```
#include <iostream>

int main() {
    char str[100];
    int num;

    std::cout << "Enter number: ";
    std::cin >> num;

    std::cout << "Enter text: ";
    cin.getline(str, sizeof(str));

    std::cout << num << " " << str;

    return 0;
}
```

The program reads the integer and stores it in num. The new line character generated when Enter is pressed is stored in the input queue. Since `getline()` terminates once the new line character is read, the user cannot enter more characters. Therefore, the program outputs only the input number. A solution is to use `cin.get()` before `getline()`, in order to read the new line character.

3.5 Pointers and Memory

Pointers are one of the most important parts of the language. It is the mechanism to access memory addresses and manipulate memory directly.

In modern computers, the main memory consists of millions of consecutively numbered memory cells where each cell stores eight bits of information and is identified by a unique number, called memory address. For example, in a computer with N bytes of memory, the memory address of each cell is a unique number from 0 to N-1, as shown here:

Memory Address	Memory Content
0	
1	
2	
⋮	
5000	10
5001	0
5002	0
5003	0
⋮	
$n-1$	

When a variable is declared, the compiler reserves the required consecutive bytes to store its value. If a variable occupies more than one byte, the variable's address is the address of the first byte.

3.5.1 Pointer Declaration

A pointer variable is a variable that can hold a memory address, such as the memory address of another variable. To declare a pointer, we write:

```
data_type *pointer_name;
```

As shown, the `*` character must precede the name of the pointer variable. For example, with the declaration: `int *ptr;`, `ptr` is declared as a pointer variable to type `int`. Therefore, `ptr` can hold the memory address of an `int` variable.

Pointer variables can be declared together with other variables of the same type. For example:

```
int *p, q, r, s;
```

Here p is a pointer to int variable, but q, r, and s are just int variables.

3.5.2 Pointer Initialization

After declaring a pointer variable, we can use it to store the memory address of another variable. To find the address of a variable we use the `&` address operator before its name. For example:

```
#include <iostream>

int main() {
    int *ptr, a;
    ptr = &a;
    std::cout << ptr << " " << &ptr << std::endl;

    return 0;
}
```

With the statement `ptr = &a;`, `ptr` becomes equal to, or "points to," the memory address of `a`. As we said, the compiler allocates memory to store the value of `ptr`. The program outputs the address of `a` and the address of `ptr`, in hexadecimal.

The assignment of an integer value to a pointer variable will probably cause a compilation error. This is logical since a pointer is not an integer. For example:

```
    int *ptr;
    ptr = 10000; // Potential compilation error.
```

3.5.3 Null Pointers

A pointer variable is initialized with a "garbage" address. To indicate that it points to nowhere, we assign `0` or the macro `NULL` (defined in headers like `cstdlib`). Such a pointer is called a *null pointer*.

For example, the following program first outputs the initial value of the pointer and then `0`:

```
#include <iostream>
#include <cstdlib>

int main() {
    int *p;
    std::cout << p << std::endl; // garbage value
    p = NULL;                    // or p = 0;
    std::cout << p << std::endl; // 0

    return 0;
}
```

3.5.4 Using Pointers

To access the content of a memory address referenced by a pointer, we use the `*` (dereference) operator. Although the same symbol is used for multiplication, the compiler determines its meaning from context.

```
#include <iostream>

int main() {
    int *ptr, a = 10;
    ptr = &a;
    std::cout << *ptr << std::endl; // prints 10
    *ptr = 20;                      // equivalent to a = 20
    std::cout << a << std::endl;    // prints 20

    return 0;
}
```

Here, `*ptr` is equivalent to the content of `a`, so the program outputs 10 and then 20.

3.5.5 Pointers and Arrays

The elements of an array are stored in successive memory locations, with the first element at the lowest memory address. The element spacing depends on the array type: in a `char` array the distance is 1 byte, while in an `int` array it equals the size of `int` (e.g., 4 bytes).

In a similar way, `arr+1` can be used as a pointer to the second element, `arr+2` as a pointer to the third one, and so on. In general:

```
  arr = &arr[0]
arr + 1 = &arr[1]
arr + 2 = &arr[2]
        ...
arr + n = &arr[n]
```

In a similar way, since `arr+1` points to the second element, `*(arr+1)` is equal to `arr[1]`, and so on. In general:

```
  *arr = arr[0]
*(arr + 1) = arr[1]
*(arr + 2) = arr[2]
        ...
*(arr + n) = arr[n]
```

Note that parentheses are required because `*` has higher precedence than `+`. Thus, `*(arr+n)` and `*arr + n` are different.

```
#include <iostream>

int main() {
    int i, arr[5] = {10,20,30,40,50};
    std::cout << "***** Using array notation *****\n";

    for(i=0;i<5;i++) {
        std::cout << "Addr: " << &arr[i] << " Val: " << arr[i] << std::endl;
    }

    std::cout << "\n***** Using pointer notation *****\n";
    for(i=0;i<5;i++) {
        std::cout << "Addr: " << arr+i << " Val: " << *(arr+i) << std::endl;
    }

    return 0;
}
```

Listing 3.2: Array and Pointer Notation

You have been provided two sample programs in **src** folder, `strings.cpp` and `pointers.cpp`. They will help you complete the exercises.

3.6 Exercises

1. [25 points] Write a C++ program `char_counter.cpp` that takes a string from the user and a character to search for, then prints the number of times the character occurs in the string.

```
>>> Enter a string: hello world
>>> Enter a character to search: l
Number of occurrences of 'l': 3
```

Hint: use a C-style string (array of characters) and iterate through it using a loop.

2. [25 points] Write a C++ program `sort_strings.cpp` that sorts an array of strings using pointers. In this program, you will practice working with arrays of characters, pointer arithmetic, and string comparison. Follow the instructions below:

   (a) **main function:**

   - First, take an integer n as input from the user. This will be the number of strings to store.
   - Create an array of size n that can store strings (character arrays).
   - Take n strings as input from the user and store them in the array.*Hint: Use* `cin.ignore()` *before reading strings with* `getline()`.
     See: https://www.javatpoint.com/cin-ignore-function-in-cpp *and* https://cplusplus.com/reference/istream/istream/ignore/
   - Pass the array of strings and its size n to the `sort` function.
   - Print the sorted array of strings.

   (b) **sort function:**

   - This function takes an array of strings and its size n as parameters.
   - Loop through the array and sort all the strings in alphabetical order.
   - *Hint: You can first use* `strcmp()` *to compare char arrays, then try to implement comparison using pointers without* `strcmp()`.

```
>>> Enter size of the array: 5
>>> Turkmenistan
>>> Uzbekistan
>>> Jamaica
>>> South Africa
>>> Guyana
Guyana
Jamaica
South Africa
Turkmenistan
Uzbekistan
```

Listing 3.3: Example

3. [25 points] Write a C++ program `array_stats.cpp` that uses pointers to process an array of numerical values. The program should:

   - Ask the user for the number of values.
   - Read the values into an array.
   - Using pointer arithmetic (not array indexing), calculate and display the following statistics:
     - Minimum value
     - Maximum value
     - Average
     - Standard deviation

```
>>> Enter number of values: 5
>>> Enter value 1: 10
>>> Enter value 2: 20
>>> Enter value 3: 30
>>> Enter value 4: 40
>>> Enter value 5: 50
Minimum: 10
Maximum: 50
Average: 30
Standard Deviation: 14.14
```

Listing 3.4: Example 1

```
>>> Enter number of values: 4
>>> Enter value 1: 5
>>> Enter value 2: 8
>>> Enter value 3: 12
>>> Enter value 4: 15

Minimum: 5
Maximum: 15
Average: 10
Standard Deviation: 4.12
```

Listing 3.5: Example 2

Hint: Use pointers to iterate through the array instead of square brackets, e.g., `*(arr+i)`.

4. [25 points] Write a function `computeStatistics` in C++ that takes as input an array of integers and its size. The function should compute the **mean**, **median**, and **mode** of the array and return them via pointers. Additionally, three boolean variables should indicate whether each value is *present* in the array.

   - **Mean:** the average of all values. It is *present* if the mean equals one of the array elements.
   - **Median:** the middle value of the sorted array. If there are two middle values, the median is their average. It is *present* if the median equals one of the array elements.
   - **Mode:** the most frequent value. If all values occur only once, there is *no mode*. If multiple values share the highest frequency, choose the largest value. It is *present* if the mode exists in the array.

```
>>> Enter number of values: 6
>>> Enter value 1: 1
>>> Enter value 2: 2
>>> Enter value 3: 3
>>> Enter value 4: 4
>>> Enter value 5: 5
>>> Enter value 6: 6
Mean: 3.5 (not present)
Median: 3.5 (not present)
Mode: None (not present)
```

Listing 3.6: Example 1

```
>>> Enter number of values: 6
>>> Enter value 1: 1
>>> Enter value 2: 2
>>> Enter value 3: 2
>>> Enter value 4: 2
>>> Enter value 5: 2
>>> Enter value 6: 3
Mean: 2 (present)
Median: 2 (present)
Mode: 2 (present)
```

Listing 3.7: Example 2

Hints:

- Sort the array first to compute the median.
- Use a frequency count to determine the mode.
- Use pointers to return the mean, median, and mode values.
- Use boolean flags to indicate whether each statistic is actually present in the array.