CS224 Object Oriented Programming and Design Methodologies

Lab Manual

Lab 03 – Function Overloading and Recursion

Department of Computer Science
Dhanani School of Science and Engineering
Habib University

Fall 2025

Contents

1 Lab Objectives

Following are the lab objectives of this lab:

- To get familiar with function overloading in C++.
- To experiment modifying the array passed to the function as a parameter, in-place.
- To get familiar with writing recursive functions in C++.
- To understand the usage of static variables in C++.

2 Sample Programs

2.1 Function Overloading Example

```cpp
#include <iostream>
using namespace std;

// Attack with just the weapon
void attack(const char* weapon) {
    cout << "Hornet strikes swiftly with her " << weapon << "!" << endl;
}

// Attack with weapon and special move
void attack(const char* weapon, const char* move) {
    cout << "Hornet uses " << move << " with her " << weapon << "!" <<
        endl;
}

// Attack with weapon and power level
void attack(const char* weapon, int power) {
    cout << "Hornet performs a mighty strike with her "
         << weapon << " and deals " << power << " damage!" << endl;
}

int main() {
    attack("Needle");                  // calls attack(const char*)
    attack("Needle", "Silk Bind");     // calls attack(const char*, const
        char*)
    attack("Needle", 5);               // calls attack(const char*, int)
    return 0;
}
```

Listing 1: function_overloading.cpp

Note that we are using the **same** function name but the function that is called is based on the parameters it is given.

If a single string is given the first "version" of the `attack` function is called. If two strings are passed the second version is called. If a string and an integer are passed then the third version is called.

The output would therefore be as follows:

```
Hornet strikes swiftly with her Needle!
Hornet uses Silk Bind with her Needle!
Hornet performs a mighty strike with her Needle and deals 5 damage!
```

Listing 2: Program Output

3 Problems

3.1 Time 24 Hours

Write three versions of a function `convert24` that converts given values of hours, minutes and seconds into a 24-hour format such as `hh:mm:ss`, where

$$0 \leq hh \leq 23, \quad 0 \leq mm \leq 59, \quad 0 \leq ss \leq 59.$$

If the given values of hours, minutes or seconds exceed these ranges, then the excess value is transformed into the next unit. For example:

- If `seconds = 65`, then this equals `1 minute and 5 seconds`.
- If `minutes = 170`, then this equals `2 hours and 50 minutes`.
- If `hours = 27`, then just print a "+1" with the balance hours, i.e. this equals `+1, 3 hours`.

The signatures of your functions are:

```
convert24(int seconds)
convert24(int minutes, int seconds)
convert24(int hours, int minutes, int seconds)
```

Listing 3: Function Signatures

Write a main function that takes as input the hours, minutes and seconds from the user and then converts and prints the time using convert24. If hours and minutes are zero use the function convert24(int seconds). Likewise if only hours is zero call convert24(int minutes, int seconds). If none of them are zero use convert24(int hours, int minutes, int seconds)

Sample Input:
Enter hours: 27
Enter minutes: 59
Enter seconds: 65

Sample Output:
+1, 04:00:05

Observe the leading zeros that need to be printed.

3.2 Complex Calculator – Overloaded

Define functions to add, subtract, and multiply complex numbers. We will use an array of two doubles to represent complex numbers: the first element will be the real part, the second is the imaginary part.

The functions should return a complex number. Since complex number operations can be combined with a real number as well, overload the functions in two variants:

- both operands are complex numbers,
- first operand is complex and second operand is a real number.

A function `show` will display the complex number in the usual way, i.e. $x + yi$.

```
add(double *complex, double *complex)
add(double *complex, double realNum)
subtract(double *complex, double *complex)
subtract(double *complex, double realNum)
multiply(double *complex, double *complex)
multiply(double *complex, double realNum)
show(double *complex)
```

Listing 4: Function Signatures

Write a main function that takes as inputs two complex numbers c1 and c2 and an integer d1. Call each of the above functions in main and display the result of each operation.

Sample Input:
Enter c1: 2 5
Enter c2: 5 3
Enter d1: 10

Sample Output:
c1+c2: $7 + 8i$
c1-c2: $-3 + 2i$
c1*c2: $-5 + 31i$
c1+d1: $12 + 5i$
c1-d1: $-8 + 5i$
c1*d1: $20 + 50i$

Explanation: First input line is c1 = 2+5i, second line is c2 = 5+3i, third line is a real number d1 = 10. The results of addition/subtraction/multiplication are given as output.

3.3 Holes in a Number/Alphabet – Overloading + Recursion

You are designing a poster that prints numbers with a different style applied to each of them. The styling is based on the number of closed paths (holes) present in the digits and letters.

Hole counts Digits:

- 1, 2, 3, 5, 7: 0 holes.
- 0, 4, 6, 9: 1 hole.
- 8: 2 holes.

Upper-case letters:

- C, E, F, G, H, I, J, K, L, M, N, S, T, U, V, W, X, Y, Z: 0 holes.
- A, D, O, P, Q, R: 1 hole.
- B: 2 holes.

Task Write a recursive function `countHoles` that takes an `int` and returns the sum of holes in its digits. Write an overloaded recursive version that takes a `char *` (a C-style string) and returns the sum of holes in its characters.

Write a main function which asks the user to enter "d" if they wish to enter a number or "s" if they wish to enter a string. Your main function should then call the respective overloaded function and outputs the number of holes.

Do not use loops; recursion must be used.

Sample Input:
1078

Sample Output:
3 holes

Sample Input:
SILKSONG

Sample Output:
1 holes

3.4 Fibonacci

Fibonacci numbers are defined as a sequence where each integer in the sequence is the sum of the previous two integers. We define the first two Fibonacci numbers as $1, 1$.

The Fibonacci sequence is:

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21, \ldots$$

(i) Write a recursive function `fib(n)` that finds and returns the nth Fibonacci number. For reference, `fib(0)` $= 1$, and `fib(1)` $= 1$.

$$\texttt{fib(2)} = \texttt{fib(1)} + \texttt{fib(0)} = 1 + 1 = 2,$$
$$\texttt{fib(3)} = \texttt{fib(2)} + \texttt{fib(1)} = 3.$$

We also want to keep track of how many times the function `fib` has been called when calculating `fib(n)`.

Write a program that prompts the user for n. Using the recursive function for `fib(n)`, it then calculates, returns, and prints `fib(n)`. Test your program for $n = 10, 20, 30, 40$, and 50. Use `long` as a return type.

How many times was `fib` called? Use a *local static variable* for this purpose.

(ii) [**Not graded**] Write an iterative version of the same program that calculates Fibonacci numbers bottom-up using a loop. Test your program for $n = 10, 20, 30, 40, 50$.