# ECE 449/590
# Homework 01
# C++ Problems

Paolo Orguim

**Due: Thursday, September 25th, 2025, 11:59 PM**

**Abstract**

This report documents the implementation of Homework 01. It includes screenshots of the outputs for various of the problems and explanations for each part of the code.

## 1 Full Code

This section contains the entirety of the code required for all the problems in the homework in their final version.

### Codes for Problems 1-5

**Listing 1:** Code for problems 1–5 (homework1_5.cpp)

```cpp
 // homework1_5.cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm> // sort, greater
#include <iterator>  // ostream_iterator, back_inserter
#include <functional>

int main() {
    std::cout << "----- Problem 1 -----\n";
    // Problem 1A: valid
    const std::string hello = "Hello";
    // hello + ", world" -> std::string, then + "!" -> std::string, valid
    const std::string messageA = hello + ", world" + "!";
    std::cout << "1A message: " << messageA << '\n';

    // Problem 1B: shown as given (invalid):
    // Sconst std::string exclaim = "!";
    // const std::string message = "Hello" + ", world" + exclaim;
    // Explanation: the expression "Hello" + ", world" is an attempt to add two C-style string pointers,
    // which is not defined. Operator+ is defined for std::string + const char* or const char* + std::string,
    // but not for two const char*.
    // Corrected forms:
    const std::string exclaim = "!";
    const std::string messageB_correct = std::string("Hello") + ", world" + exclaim;
    std::cout << "1B corrected message: " << messageB_correct << '\n';

    std::cout << "\n----- Problem 2 -----\n";
    // Problem 2: assignment associativity and behavior
```

```
    int a(0), b(1), c(2), d(3);
    // For the chain a = b = c = d; assignment must be right-associative:
    // equivalent to a = (b = (c = d));
    // Each assignment sets the left operand equal to the right operand.
    a = b = c = d;
    std::cout << "a b c d after chaining: " << a << " " << b << " " << c << " " << d << '\n';

    std::cout << "\n----- Problem 3 -----\n";
    // Problem 3A: incorrect method
    std::vector<int> u(10, 100); // 10 elements all 100
    std::vector<int> v;          // empty

    // std::copy(u.begin(), u.end(), v.begin());

    std::cout << "3A: Attempted incorrect std::copy from u to v (commented in code).\n";
    std::cout << "Reason: v.begin() is not a valid output range because v has size 0.\n";

    // Problem 3B correct way:

    // Direct copy assignment
    std::vector<int> v_fix = u;
    std::cout << "3B fix (direct assignment) v_fix size: " << v_fix.size() << ", elements: ";
    for (auto x : v_fix) std::cout << x << " ";
    std::cout << "\n";

    std::cout << "\n----- Problem 4 -----\n";
    // Problem 4: iterate with iterator to print each element
    std::vector<int> temp = {1,2,3,4,5};
    std::cout << "Elements of temp using iterator: ";
    for (std::vector<int>::const_iterator it = temp.begin(); it != temp.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";

    std::cout << "\n----- Problem 5 -----\n";
    // Problem 5: sort integers from largest to smallest
    std::vector<int> sample = {4, 1, 9, 7, 3, 8, 2};
    std::cout << "Before sort (sample): ";
    for (auto x : sample) std::cout << x << " ";
    std::cout << "\n";

    // sort descending using std::greater<>
    std::sort(sample.begin(), sample.end(), std::greater<int>());

    std::cout << "After sort (descending): ";
    for (auto x : sample) std::cout << x << " ";
    std::cout << "\n";

    return 0;
}
```

## Code for Problem 6

**Listing 2:** Code for problem 6 (hw1_q6.cpp)

```
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
#include <chrono>
int main()
```

```cpp
{
    // Empty vector and doubly-linked list
    std::vector<int> integers_vector;
    std::list<int> integers_list;

    // Max # of elements
    size_t max_size = 1000;

    std::cout << "inserting values into vector and list..." << std::endl;
    for (size_t i = 0; i < max_size; i++)
    {
        integers_vector.push_back(i);
        integers_list.push_back(i);
    }

    // Choose a random number
    size_t random_number = rand() % max_size + 1;
    std::cout << "random number to find in vector and list is: " << random_number << std::endl;

    // Time the search on the vector
    std::cout << "searching in vector..." << std::endl;
    auto start_vector = std::chrono::high_resolution_clock::now();
    std::find(integers_vector.begin(), integers_vector.end(), random_number);
    auto end_vector = std::chrono::high_resolution_clock::now();

    // Time search on the list
    std::cout << "searching in list..." << std::endl;
    auto start_list = std::chrono::high_resolution_clock::now();
    std::find(integers_list.begin(), integers_list.end(), random_number);
    auto end_list = std::chrono::high_resolution_clock::now();

    // Durations in milliseconds
    std::chrono::duration<double, std::milli> vector_time = end_vector - start_vector;
    std::chrono::duration<double, std::milli> list_time = end_list - start_list;

    // Results
    std::cout << "Time took searching " << random_number << " in vector: " << vector_time.count() << "ms" << std::endl;
    std::cout << "Time took searching " << random_number << " in list: " << list_time.count() << "ms" << std::endl;

    return 0;
}
```

## 2 Problem 1

Only the definition on part A is valid. As we can see from the screenshot in Figure 2 we cannot use a + operator on two strings, this operation is not defined and an error comes up during compilation.

### Part A

```
----- Problem 1 -----
1A message: Hello, world!
```

**Figure 1:** Result from running the original code for message A.

### Part B

```
ece449@ECE449:~/Desktop/my449$ g++ -std=c++17 -O2 -Wall -o hw1 homework1_5.cpp
homework1_5.cpp: In function 'int main()':
homework1_5.cpp:19:41: error: invalid operands of types 'const char [6]' and 'const char [8]' to binary 'operator
+'
   19 |     const std::string message = "Hello" + ", world" + exclaim;
      |                                 ~~~~~~~ ^ ~~~~~~~~~
      |                                 |       |
      |                                 |       const char [8]
      |                                 const char [6]
```

**Figure 2:** Result from running the original code for message B.

## 3 Problem 2

In order for the output to be `3 3 3 3` we need right-to-left associativity, that means that all the variables will have the same value as the `d` variable (the right-most one). We can see that is the case in Figure 3.

```
std::cout << "\n----- Problem 2 -----\n";
// Problem 2: assignment associativity and behavior
int a(0), b(1), c(2), d(3);
// For the chain a = b = c = d; assignment must be right-associative:
// equivalent to a = (b = (c = d));
// Each assignment sets the left operand equal to the right operand.
a = b = c = d;
std::cout << "a b c d after chaining: " << a << " " << b << " " << c << " " << d << '\n';
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS bash

```
----- Problem 2 -----
a b c d after chaining: 3 3 3 3
```

**Figure 3:** Code and output from running it.

## 4 Problem 3

### Part A

It is an incorrect method because `v.begin()` is an invalid range since the vector `v` is empty.

### Part B

A correct implementation of this exercise is shown below in Figure 4.

```
    std::cout << "\n----- Problem 3 -----\n";
    // Problem 3A: incorrect method
    std::vector<int> u(10, 100); // 10 elements all 100
    std::vector<int> v;          // empty

    // std::copy(u.begin(), u.end(), v.begin());

    std::cout << "3A: Attempted incorrect std::copy from u to v (commented in code).\n";
    std::cout << "Reason: v.begin() is not a valid output range because v has size 0.\n";

    // Problem 3B correct way:

    // Direct copy assignment
    std::vector<int> v_fix = u;
    std::cout << "3B fix (direct assignment) v_fix size: " << v_fix.size() << ", elements:
    for (auto x : v_fix) std::cout << x << " ";
    std::cout << "\n";
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS bash

```
----- Problem 3 -----
3A: Attempted incorrect std::copy from u to v (commented in code).
Reason: v.begin() is not a valid output range because v has size 0.
3B fix (direct assignment) v_fix size: 10, elements: 100 100 100 100 100 100 100 100 100 100
```

**Figure 4:** Code and output from running it.

# 5 Problem 4

Below, in Figure 5, is the implementation of the iterator as well as the output from running the code.

```
    std::cout << "\n----- Problem 4 -----\n";
    // Problem 4: iterate with iterator to print each element
    std::vector<int> temp = {1,2,3,4,5};
    std::cout << "Elements of temp using iterator: ";
    for (std::vector<int>::const_iterator it = temp.begin(); it != temp.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS bash

```
----- Problem 4 -----
Elements of temp using iterator: 1 2 3 4 5
```

**Figure 5:** Code and output from running the code.

# 6 Problem 5

In Figure 6 we can see the code for the sorting program as well as the output from running it.

```
    std::cout << "\n----- Problem 5 -----\n";
    // Problem 5: sort integers from largest to smallest
    std::vector<int> sample = {4, 1, 9, 7, 3, 8, 2};
    std::cout << "Before sort (sample): ";
    for (auto x : sample) std::cout << x << " ";
    std::cout << "\n";

    // sort descending using std::greater<>
    std::sort(sample.begin(), sample.end(), std::greater<int>());

    std::cout << "After sort (descending): ";
    for (auto x : sample) std::cout << x << " ";
    std::cout << "\n";

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
----- Problem 5 -----
Before sort (sample): 4 1 9 7 3 8 2
After sort (descending): 9 8 7 4 3 2 1
```

**Figure 6:** Code and output from running it.

# 7 Problem 6

## Part A

Comments are shown in the code on section 1.

## Part B

20 random lookup trials were run for each container size and averaged the times. The overall average find time across the six sizes was about 3.03 ms for `std::vector` and 18.52 ms for `std::list`. The vector is consistently faster and the gap grows with container size. This happens because `std::vector` stores elements in contiguous memory, which makes sequential scans take advantage of CPU caches and hardware prefetching. `std::list` stores each element in a separate node and requires pointer chasing, which causes many cache misses and higher memory-access latency. For very small containers the times are both near zero and differences are negligible, but for large sizes the cache-locality advantage of `vector` dominates.

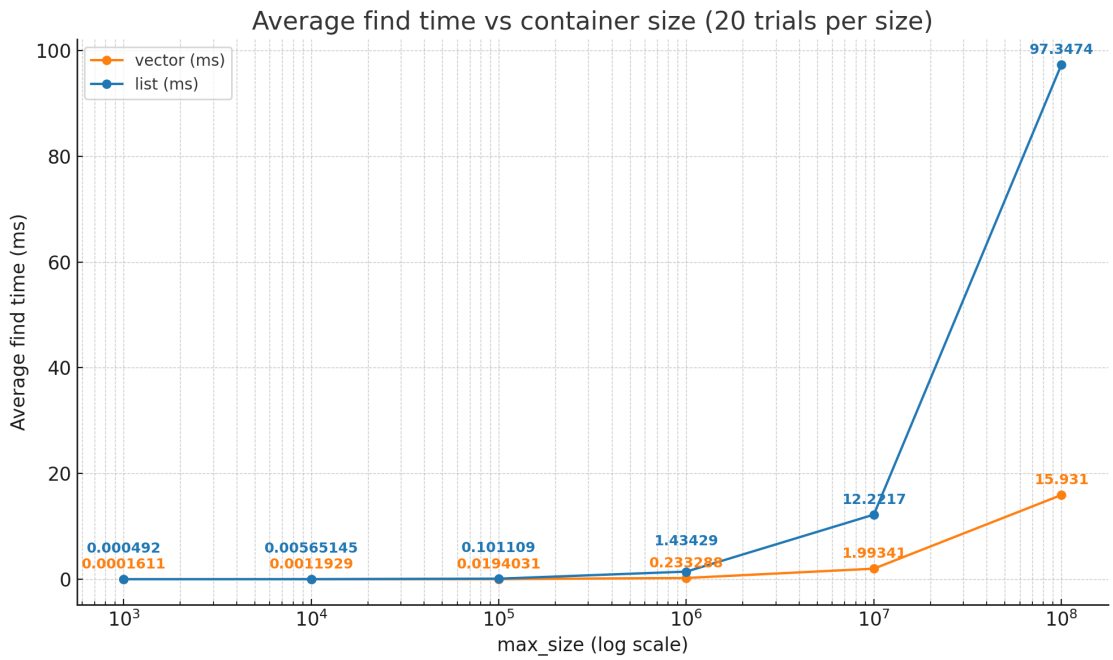


**Figure 7:** Graph comparing times between vector and list implementations.

The code used to make the benchmark is shown below, it is important to notice that both `-O3` and `-march=native` flags were used to compile the program.

**Listing 3:** Code for problem 6 benchmark

```
// hw1_6_benchmark.cpp
#include <iostream>
#include <algorithm>
#include <list>
#include <vector>
#include <chrono>
#include <random>
#include <numeric>
#include <cstdint>

int main()
{
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);

    // sizes to test (user requested)
    const std::vector<std::size_t> sizes = {
        100'000'000ULL,
         10'000'000ULL,
          1'000'000ULL,
            100'000ULL,
             10'000ULL,
              1'000ULL  // ' // Comment to fix highlighting in latex
    };

    const int trials = 20; // number of lookups to average per size

    // RNG for choosing random lookup target
    std::mt19937_64 rng((unsigned)std::chrono::high_resolution_clock::now().time_since_epoch
    ().count());

    // Accumulators for overall averages (only include sizes where list was built)
```

```
    double overall_vector_sum_ms = 0.0;
    double overall_list_sum_ms = 0.0;
    std::size_t overall_vector_counts = 0;
    std::size_t overall_list_counts = 0;

    for (std::size_t max_size : sizes) {
        std::cout << "====================\n";
        std::cout << "Testing max_size = " << max_size << "\n";

        // Estimate memory used by a std::list node: int + two pointers
        // (this is an approximation; real node size depends on implementation)
        std::uint64_t approx_node_bytes = sizeof(int) + 2 * sizeof(void*);
        long double estimated_list_bytes = (long double)approx_node_bytes * (long double) max_size;
        long double estimated_list_mb = estimated_list_bytes / (1024.0L * 1024.0L);

        std::cout << "Estimated list memory (approx): " << estimated_list_mb << " MB\n";

        // Heuristic safety threshold: skip building list if estimated > 2048 MB (2 GB)
        const long double LIST_MEMORY_THRESHOLD_MB = 2048.0L;

        bool build_list = (estimated_list_mb <= LIST_MEMORY_THRESHOLD_MB);
        if (!build_list) {
            std::cout << "WARNING: estimated list memory > " << LIST_MEMORY_THRESHOLD_MB << " MB. ";
            std::cout << "Skipping construction of std::list for this size to avoid OOM.\n";
        }

        // Containers
        std::vector<int> integers_vector;
        std::list<int> integers_list;

        integers_vector.reserve(max_size); // avoid reallocation noise for vector

        std::cout << "Filling containers...\n";
        for (std::size_t i = 0; i < max_size; ++i) {
            integers_vector.push_back(static_cast<int>(i));
            if (build_list) integers_list.push_back(static_cast<int>(i));
        }

        // prepare random distribution for lookups in [0, max_size-1]
        std::uniform_int_distribution<std::size_t> dist(0, max_size - 1);

        // store trial timings
        std::vector<double> vector_times_ms;
        std::vector<double> list_times_ms;
        vector_times_ms.reserve(trials);
        if (build_list) list_times_ms.reserve(trials);

        std::cout << "Running " << trials << " lookup trials...\n";
        for (int t = 0; t < trials; ++t) {
            std::size_t target = dist(rng);

            // measure vector find
            auto sv = std::chrono::high_resolution_clock::now();
            auto itv = std::find(integers_vector.begin(), integers_vector.end(), static_cast<int>(target));
            auto ev = std::chrono::high_resolution_clock::now();
            double v_ms = std::chrono::duration<double, std::milli>(ev - sv).count();
            vector_times_ms.push_back(v_ms);
            bool found_in_vector = (itv != integers_vector.end()); // use result

```

```
        // measure list find if built
        if (build_list) {
            auto sl = std::chrono::high_resolution_clock::now();
            auto itl = std::find(integers_list.begin(), integers_list.end(), static_cast<int>(target));
            auto el = std::chrono::high_resolution_clock::now();
            double l_ms = std::chrono::duration<double, std::milli>(el - sl).count();
            list_times_ms.push_back(l_ms);
            bool found_in_list = (itl != integers_list.end());

            if (!found_in_vector || !found_in_list) {
                std::cerr << "ERROR: element not found in one of the containers (this should not happen)\n";
            }
        } else {
            // still sanity-check vector only
            if (!found_in_vector) {
                std::cerr << "ERROR: element not found in vector (this should not happen)\n";
            }
        }
    } // end trials

    // compute per-size averages
    double avg_v = std::accumulate(vector_times_ms.begin(), vector_times_ms.end(), 0.0) / vector_times_ms.size();
    std::cout << "Average vector find time over " << vector_times_ms.size() << " trials: " << avg_v << " ms\n";

    overall_vector_sum_ms += avg_v;
    overall_vector_counts++;

    if (build_list) {
        double avg_l = std::accumulate(list_times_ms.begin(), list_times_ms.end(), 0.0) / list_times_ms.size();
        std::cout << "Average list find time over " << list_times_ms.size() << " trials: " << avg_l << " ms\n";
        overall_list_sum_ms += avg_l;
        overall_list_counts++;
    } else {
        std::cout << "List timings skipped for this size.\n";
    }

    // free memory before next iteration
    integers_vector.clear();
    integers_vector.shrink_to_fit();
    if (build_list) {
        integers_list.clear();
    }

    std::cout << "Done with size " << max_size << "\n\n";
} // end sizes loop

// print overall averages (across sizes that were tested)
if (overall_vector_counts > 0) {
    std::cout << "=== Overall average across " << overall_vector_counts << " sizes (vector) ===\n";
    std::cout << (overall_vector_sum_ms / (double)overall_vector_counts) << " ms\n";
}
if (overall_list_counts > 0) {
    std::cout << "=== Overall average across " << overall_list_counts << " sizes (list) ===\n";
```

```
        std::cout << (overall_list_sum_ms / (double)overall_list_counts) << " ms\n";
    } else {
        std::cout << "No list timings were produced (list was skipped for very large sizes).\n";
    }

    return 0;
}
```