CS101A(H) Homework One

Brief Overview

This homework consists of four parts:

- **Multiple Choices (20 pts):** A set of multiple-choice questions covering basic C++ concepts.
- **Codes (23 pts):** Mandatory programming exercises. You must implement and demonstrate the required functions in `mini_data_analyzer.cpp`.
- **Check (9 pts):** This part will be done after the discussion class in person. In the homework description you may notice many "**why**" annotations. The goal of the Check is to ensure that you truly understand every option and concept, rather than simply choosing an answer. During Check, we will randomly select **three questions or explanations** from the homework (including the "why" parts), and you will be asked to explain them. Each explanation is worth 3 points. This also encourages you to consult references and fully digest the reasoning provided in the assignment.
- **Optional Extensions (6 pts):** Extra coding challenges for additional practice. These are not required but recommended if you want to deepen your understanding. Optional Extensions *do not mean they will never appear in your midterms or finals*; in fact, they might. This part emphasizes finer details of the lecture and is *not beyond the syllabus*. They provide more interesting ways of thinking about the class material. Therefore, if time permits, we strongly recommend that you attempt them.

Your grade for this homework will be computed as

$$\min\Big(\text{Multiple Choices} + \text{Codes} + \text{Check},\ 50\text{ pts}\Big) + \text{Optional Extensions}.$$

Notes: Before starting this homework, please note that we assume you already possess some basic knowledge of C++ programming. If you are still unfamiliar with how a C++ program works, please refer to: `https://www.learncpp.com/`. You should also understand some fundamental concepts such as control flow, functions, etc. This week's homework is designed under the assumption that you have at least some prior experience with C++. Building on the content from the first week's lecture, this homework will cover basic programming tasks and exercises, including variables, data types, control structures, pointers, arrays, reference, vectors, etc. For further reference, please consult: `https://en.cppreference.com/w/`, `https://cplusplus.com/doc/tutorial/`.

Usage of AI. The use of AI tools for searching information or obtaining assistance on this homework is strictly prohibited. All solutions must be derived from your own understanding and effort. Submissions will be reviewed carefully, and any indication of reliance on AI-generated content will result in severe penalties.

Academic Integrity. This course upholds the highest standards of academic integrity. Any form of academic dishonesty, including plagiarism, unauthorized collaboration, or the use of prohibited resources such as AI tools, will be treated as a serious violation. Such actions not only undermine your own learning but also violate university policies.

1 Multiple Choices

1. Select the correct statement(s). (2 pts)

A. We can set up our own IDE using VSCode with extensions, compilers, and debuggers.
B. C++23 is the latest standard (2023). To use it, we need a compiler supporting C++23.
C. GCC only supports Windows; on Linux we must use Clang.
D. A C++ program can be executed directly without being compiled into an executable.
E. We can compile and run C/C++ programs in the terminal.
F. VSCode and CLion are different types of compilers.

Q1 Answer and Explanation **Answer : A, B, E**

- A: Correct. VSCode can be set up as an IDE with compiler/debugger extensions.
- B: Correct. To use C++23, the compiler must support the standard.
- E: Correct. C/C++ programs can be compiled and executed in the terminal.
- C: Incorrect. GCC is widely used on Linux/Unix and other platforms, not only Windows.
- D: Incorrect. A C++ program must be compiled before execution.
- F: Incorrect. VSCode/CLion are IDEs, not compilers.

2. For each type, indicate its signedness. (2 pts)
Choose the correct option (A–C) for each row in the table.

A. signed B. unsigned C. implementation-defined

Type	Your Choice (A–C)
char	_____
signed char	_____
unsigned char	_____
short	_____
unsigned	_____
int	_____
long	_____
long long	_____

Answer:

Type	**Your Choice**
char	C
signed char	A
unsigned char	B
short	A
unsigned	B
int	A
long	A
long long	A

3. Which of the following snippets involve integer *overflow*? Assume `int` is 32-bit, `long long` is 64-bit. (2 pts)

A. `unsigned u1 = 10000001; u1 = u1*u1*u1*u1;`

B. `int inf = 42 / 0;`

C. `long long ival = -100000000000000; unsigned uval = ival;`

D. `int ival = 1000000; long long llval = ival * ival;`

Answer: D

- A: Unsigned integer doesn't overflow.
- D: `int` multiplication happens first; $10^6 \times 10^6 = 10^{12}$ exceeds 32-bit signed range → signed overflow = UB.
- B: Division by zero is UB, but not overflow.
- C: Converting signed to unsigned is well-defined (mod 2^N), not overflow.

4. Which of the following code snippets invoke *undefined behavior* (UB, see cppreference for reference)? Assume `int` is 32-bit. (2 pts)

A. `unsigned uval = -111; printf("%u", uval);`

B. `int x = 96; printf("%f", x/100);`

C.
```
int helper() {
    int x;
    return helper() + x;
}
```

D.
```
int f() {
    int y;
    return y;
}
```

Answer: B, C, D

- B: Wrong format specifier (%f expects double, given int) → UB.
- C: `++i` and `i` used in the same expression without sequence point → UB.
- D: Recursive call uses uninitialized variable `x` → UB.
- A: Assigning negative to unsigned is well-defined (mod 2^{32}), not UB.

5. Arrays and pointers. (2 pts) Given

```
int a[] = {1,2,3};
int b[] = {100,101,102};
int c[100] = {1};
```

Which statements are true?

A. The type of `a` is `int[3]`.

B. We can assign `b = a;` to make `b` point to `a`.

C. After initialization, only `c[0]` is 1, all other elements of `c` are 0.

D. Array `c` has 101 elements.

Answer: A, C

- A: Correct. The type of `a` is indeed `int[3]`.
- C: Correct. `int c[100] = {1};` zero-initializes the rest after `c[0]`.
- B: Incorrect. Arrays are not assignable; `b = a;` is ill-formed.
- D: Incorrect. `c` has exactly 100 elements, not 101.

6. Functions and pointers. (2 pts) Given

```
void swap(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

Which statements are true?

A. The code swaps the values of `*pa` and `*pb`.

B. Changing declaration to `void swap(int a, int b)` still swaps `x` and `y` (suppose we pass `x, y` to `swap`).

C. The declaration `int *pa, pb;` makes both `pa` and `pb` pointers.

D. Passing `&x, &y` to `swap` swaps `x` and `y`.

Answer: A, D

- A: Correct. `swap` exchanges the values stored at the two addresses.
- D: Correct. Passing `&x, &y` swaps the actual objects `x` and `y`.
- B: Incorrect. `swap(int a, int b)` uses pass-by-value and does not affect callers' variables.
- C: Incorrect. In `int *pa, pb;`, only `pa` is a pointer; `pb` is an `int`.

7. min_element. (2 pts) Consider

```
int min_element(int *arr, int l, int r) {
  int pos = l;
  while (l < r) {
    if (arr[l] < arr[pos]) pos = l;
    ++l;
  }
  return pos;
}
```

Which statements are true?

A. The function returns the index of the *first* minimum element in [l,r).

B. To return the *last* occurrence of the minimum value, change the comparison in `if (arr[l] < arr[pos])` to `if (arr[l] <= arr[pos])`.

C. Passing an *invalid range l,r* —i.e., any range violating $0 \le l < r \le n$ (where n is the length of `arr`)—will always cause undefined behavior.

Answer: A, B

- A: With `if (arr[l] < arr[pos])`, ties do not update `pos`, so it returns the *first* minimum.

- B: Changing to `if (arr[l] <= arr[pos])` updates on ties, yielding the *last* minimum.

8. sort using swap & min_element. (2 pts)
Consider the following implementation of selection sort:

```
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

int min_element(int* arr, int l, int r) {
  int pos = l;
  while (l < r) {
    if (arr[l] < arr[pos]) pos = l;
    ++l;
  }
  return pos;
}

void sort(int* arr, int n) {
  for (int i = 0; i < n - 1; ++i) {
    int pos = min_element(arr, i, n);
    swap(&arr[i], &arr[pos]);
  }
}
```

Which statements are true?

A. At each iteration, the smallest element in the suffix is placed at index `i`.

B. The function sorts the array in non-decreasing order.

C. Modifying loop bound from `i<n-1` to `i<n` does not change correctness.

Answer: A, B, C

- A: Each iteration places the smallest element of the suffix `[i, n)` at index `i` (selection step).
- B: Repeating the selection step from `i=0` to `n-2` yields non-decreasing order.
- C: Using `i < n` instead of `i < n-1` only adds a redundant final pass (swap with itself); correctness unchanged.

9. Pointer difference. (2 pts)
Given

```
int z[8]{0,1,2,3,4,5,6,7};
int* p = &z[6];
int* q = &z[1];
```

Which of the following statements are *correct*?

A. The expression `p - q` is well-defined and its value is `5`.

B. The type of `(p - q)` is `std::ptrdiff_t` (a signed integer type).

C. The expression `q - p` is well-defined and its value is `-5`.

D. The expression `q - (z + 9)` is well-defined and equals `-8`.

E. The expression `(z + 8) - z` is well-defined and equals `8`.

Answer: A, B, C, E

- A: Correct. `p` points to `z[6]`, `q` points to `z[1]`. The difference `p - q` is `6 - 1 = 5`.
- B: Correct. The result of subtracting two pointers is of type `std::ptrdiff_t`, which is a signed integer type.
- C: Correct. `q - p` is `1 - 6 = -5`.
- D: Incorrect. `z + 9` points to one element past the end of the array of 8 elements (`z[8]`), which is a valid "one-past-the-end" pointer.
- E: Correct. `z + 8` is the "one-past-the-end" pointer of the array `z`. Subtracting the pointer to the first element (`z`) from it yields the size of the array, which is `8`. This is well-defined.

10. Sequencing Pitfalls and Safe Rewrites. (2 pts)

Consider the following four code snippets in C++:

```
i = i++ + ++i;
v[i] = i++;
f(g(), h());
a[i] > 0 && i > 0;
```

Which of the following statements are **correct**? (Multiple answers may be correct.)

A. `i = i++ + ++i;` has undefined behavior.

B. `v[i] = i++;` is well-defined: the array index is evaluated before the post-increment, so it is safe.

C. `f(g(), h());` may have undefined behavior because the evaluation order of `g()` and `h()` is unspecified.

D. `a[i] > 0 && i > 0;` is safe, because `&&` guarantees left-to-right evaluation.

Answer: A, C

- A: Correct. `i = i++ + ++i;` performs two unsequenced modifications/uses of `i` within the same full expression (one post–increment and one pre–increment). Because there's no sequencing relation between them, this is undefined behavior.
- B: Incorrect. In `v[i] = i++;` the evaluation of the subscript `i` and the post–increment `i++` are unsequenced relative to each other. This yields an unsequenced read/modify of the same scalar `i`, which is undefined behavior. So it is *not* well-defined.
- C: Correct. In `f(g(), h());` the order of evaluating `g()` and `h()` is unspecified. If these calls have side effects on the same object that require a sequencing relation (e.g., both read/modify the same global without synchronization), the program *may* exhibit undefined behavior. Hence the statement—"may have UB because the order is unspecified"—is correct.
- D: Incorrect. Although `&&` guarantees left-to-right evaluation and sequences the right operand after the left, `a[i] > 0 && i > 0;` is not *in general* safe: `a[i]` is evaluated *before* checking

`i > 0`, so if `i` is out of bounds or negative, the left operand can already cause undefined behavior from an invalid array access. Therefore calling it "safe" is wrong.

2 Codes

Build: `make public` **Run:** `./build/public_test`

Submission model. Please submit the relevant .hpp files for each problem to the OJ. Implement all required functions in this single header. Do *not* provide a `main()`; the supplied testers (public and grader) will include your header and exercise your code.

Structure. Implement each milestone as a standalone function that *exactly* matches the given prototypes. The provided testers will print labeled sections for each milestone. Do *not* use `using namespace std;` at global scope. **(Why?)**

- **Namespace pollution.** Pulling all names from `std` into the global namespace risks accidental clashes with your own identifiers (e.g., `count`, `distance`, `size`, `swap`) and with third-party libraries.
- **Name collisions & ambiguous overloads.** Common function names (`begin`/`end`/`swap`) may become ambiguous or silently bind to an unintended overload, producing brittle code that changes behavior after a header include order change.
- **Safer alternatives.** Prefer (1) fully qualified names (`std::vector`, `std::string`); (2) *narrow* using-declarations inside a function or block (`using std::cout; using std::string;`); (3) namespace aliases for long names (`namespace rng = std::ranges;`).

A. Pointers & Arrays

A.1 **Pointer-only scans and transforms (4 pts).**

Task: Implement three fundamental buffer operations using *only pointer arithmetic* (no `[]`) and *O(1)* extra space. All functions treat the input as a contiguous block `int buf[N]`.

```
int  count_positive(const int* p, int N);
void reverse_inplace(int* p, int N);
```

Description of each function:

- `count_positive`: returns the number of strictly positive elements in the range `[p, p+N)`.
- `reverse_inplace`: reverses the entire range `[p, p+N)` in place (e.g., `[1,2,3]`→`[3,2,1]`). Do nothing when $N \leq 1$.

Constraints. No subscripting; only pointer increments/decrements, comparison, dereference. No extra arrays or vectors (only a few scalars allowed).

A.2 **2D array processing without `int**`.(6 pts)**

In C++, a **template** is a way to write code that works for many types or constants without repeating yourself.

- **Function templates** allow the type to be a parameter. Example:
  ```
  template <class T>
  T add(T a, T b) { return a + b; }
  ```
  Here the compiler deduces `T` when you call `add(3,4)` or `add(1.5,2.5)`.
- **Non-type template parameters** allow a compile-time constant (like an array size) to

be part of the template. Example:

```
template <size_t N>
void printRow(const int (&row)[N]) {
    for (size_t j = 0; j < N; ++j) std::cout << row[j] << ' ';
}
```

Here `N` is known at compile time, and the compiler can check array bounds.

Why is the template useful? You may find these links helpful: cppreference , GeeksforGeeks , LearnCpp

Task: You need to fill in the functions below. `sum2d_knownC` sums all elements when the number of columns is known at compile time but rows are given at runtime. `sum2d_ref` preserves both dimensions in the type. `transpose` writes $B[j][i] \leftarrow A[i][j]$.

```
template <size_t C>
int sum2d_knownC(int (*a)[C], int R);   // pointer-to-array, C is compile-time

template <size_t R, size_t C>
int sum2d_ref(const int (&a)[R][C]);    // reference to real 2D array

template <size_t R, size_t C>
void transpose(const int (&A)[R][C], int (&B)[C][R]);
```

Reminder: Do not use `int**`. **Why not `int**`?**

- **True shape preservation.** A 2D array `int a[R][C]` is a contiguous block in memory. Passing it as `int**` loses the compile-time column information; templates like `int (&a)[R][C]` preserve both dimensions in the type.

B. References & Vectors

B.1 Uppercasing: pointer-range vs reference. (4 pts)

```
void to_upper_ptr(char* first, char* last); // modifies [first,last)
void to_upper_ref(std::string& s);          // modifies via reference
```

Task:

- `to_upper_ptr`: Walks from `first` up to but not including `last`, uppercasing each character in place. Typical call: `to_upper_ptr(&s[0], &s[0] + s.size());`.
- `to_upper_ref`: Iterates the `std::string` by reference (`for (char& c : s)`) and uppercases each element.

Hints.

- Use `std::toupper` from `<cctype>`. See the reference: cppreference: std::toupper.
- Always cast to `unsigned char` before calling `std::toupper` to avoid undefined behavior: `c = static_cast<char>(std::toupper(static_cast<unsigned char>(c)));`
- **Why is the cast to `unsigned char` needed here instead of using a plain `char`?** The function `std::toupper` expects its argument to be either `EOF` or an `unsigned char` value in the range $[0, 255]$. If a plain signed `char` with a negative value is passed (possible on many platforms), the behavior is undefined. Casting to `unsigned char` ensures the value is always in the valid range and avoids UB.
- Demonstrate in `main()` that looping `for (char c : s)` (by value) does not mutate the

string, while `for (char& c : s)` does. **Why?** In the loop `for (char c : s)`, each character is copied into a temporary variable `c`, so modifications affect only the copy, not the string. In contrast, `for (char& c : s)` binds `c` as a reference to each element in the string, so any assignment updates the original characters in place.

B.2 **Vector tail cleanup & safe access. (2 pts)**

```
void trim_trailing_evens(std::vector<int>& v);
```

Task:

- `trim_trailing_evens`: While the vector is non-empty and the last element is even, call `v.pop_back()`.
- Implement `print_index_if`: If index is in range, print `v[i]`. Otherwise, print a custom message. Note that `operator[]` itself does not check bounds, so you must check manually.

C. Safe Values & Expressions

C.1 **Limits & wrap-around demo (no UB). (4 pt)**

Task: Print `int32_t` min/max and `uint32_t` max. Show that `uint32_t` wraps modulo 2^{32} (e.g., `UINT32_MAX + 1 == 0`). Note: signed overflow is undefined behavior; do not rely on wrap for signed types.

C.2 **Safe addition helper. (3 pt)**

```
bool add_safe(int x, int y, int& out);
```

Task: Compute $x + y$ in wider integer type, check against `int` bounds, store in `out` and return `true` if safe; otherwise return `false` and leave `out` unchanged.

3 Optional Extensions for Codes

O1. Array ↔ Vector Bridge & Pointer Invalidation (3 pt)

Starting from `int a[6]{1,2,3,4,5,6};`:

- Build `std::vector<int> v(a, a+6)` and obtain a raw pointer via `int* p = v.data();`.
- Mutate the elements through `p` (e.g., negate every value) and print the vector after mutation.
- Demonstrate pointer invalidation by growing the vector until a reallocation occurs (e.g., `push_back` in a loop) and print whether `v.data()` has changed.
- Show a safe alternative: either `reserve` enough capacity before growth, or *re-fetch* `v.data()` after growth before using the pointer again.

O2. API Design Reflection (3 pt)

Print a short, labeled paragraph (via `std::cout`) reflecting on:

- When pointer parameters are preferable.
- When references are better.

Include one concrete example of each in prose or code comments.