

Tutorial 4: Memory Management, C/C++ Collections, and Amortized Complexity

The Art of Computer Programming

7 October 2026

Objective. This tutorial puts into practice the fourth lecture: arrays and pointers in C, the regions of memory of a program, dynamic allocation with `malloc` and `free` and the errors that come with it, the collections of the C++ standard library, and the amortized complexity of insertion in a dynamic array. Do Parts A and B (C arrays, the heap) in the first half of the session and Parts C and D (the same programs in C++ with `std::vector`, `std::string`, and `std::map`; dynamic arrays) in the second half; Part E is for those who finish early. Exercises are roughly ordered by importance within each part.

Reminders

- Create a folder `tutorial04`, one file per exercise. Compile C programs with `$gcc -Wall -Wextra -std=c23 arrays.c -o arrays` and C++ programs with `$g++ -Wall -Wextra -std=c++17 vectors.cpp -o vectors`.
- `size_t` is the unsigned integer type of sizes and lengths, returned by `sizeof`, `strlen`, and `v.size()`, and printed with `%zu`; declare an index compared with such a value as `size_t` too, or `-Wall -Wextra` warns about comparing signed and unsigned integers.
- *The memory checker.* Adding `-g -fsanitize=address,undefined` to the `gcc` or `g++` command line compiles the program with AddressSanitizer: the program then stops at the first memory error with a report naming the kind of error and the line, and lists the blocks never freed when it exits (not on macOS). Use it whenever your compiler supports it; the GCC of MSYS2 on Windows does not: do Q5 with a neighbor, or paste your program into Compiler Explorer (<https://godbolt.org/>, same flags). Never time a program compiled with the checker, which slows it down several times.
- Reading input in C++: `int n; std::cin >> n;` reads an integer, `std::string w; std::cin >> w;` reads a word, and `while (std::cin >> w)` loops until the end of the input. To feed a file to a program, run `./freq < alice.txt` (zsh, bash) or `$Get-Content alice.txt | .\freq.exe` (PowerShell).
- Timing and the doubling test: see the reminders of tutorial 3; in C++, `clock()` is in `<ctime>`.

Part A: Arrays and pointers in C

Q1. Predict before running. Write down what the following program prints, line by line, then compile it, read the warning, and run it to check. Explain each line using pointer arithmetic, the decay of an array to a pointer, and `sizeof`. What happens when the two commented-out lines are restored?

```
#include <stdio.h>

void f(int t[]) {
    printf("%zu\n", sizeof(t) / sizeof(t[0]));
    t[0] = 100;
}

int main(void) {
    int a[] = {2, 3, 5, 7, 11};
    int *p = a + 1;
    printf("%zu\n", sizeof(a) / sizeof(a[0]));
```

```
    printf("%d %d %d\n",
           a[2], *(a + 2), p[1]);
    printf("%d %d\n", *p + 1, *(p + 1));
    p++;
    printf("%d %d\n", *p, p == &a[2]);
    f(a);
    printf("%d\n", a[0]);
    const int b[] = {1, 2};
    // b[0] = 3;
    // a = b;
}
```

Q2. Functions on arrays. In a file `arrays.c`, write the following functions, and test each of them in `main` on the array `int a[] = {4, 8, 15, 16, 23, 42, 8}`, computing its length with `sizeof` rather than writing 7.

- `int max_array(const int *a, int n)`, returning the largest element.
- `void reverse(int *a, int n)`, reversing the array in place (swap `a[i]` and `a[n - 1 - i]`, tutorial 2). Why is `reverse` the only function whose first parameter is not `const`? Add `const` to it and read the compiler's error.
- `void print_array(const int *a, int n)`, printing the elements on a line, to check `reverse`.
- Why do these functions need the parameter `n`, where Python's `max(1)` does not? Call `max_array` with 20 instead of the length: does the compiler complain? Does the program? Keep this in mind for Part B.

Q3. Two-dimensional arrays. Declare `int m[3][4]` and fill it with `m[i][j] = 10 * i + j` using two nested loops.

- Write `void print_matrix(int m[][4], int rows)`, printing one row per line. Why must the number of columns appear in the type of the parameter, but not the number of rows?
- Write `void transpose(int m[][4], int t[][3])` filling `t` with the transpose of `m` (rows and columns exchanged) and print `t` with a second printing function. Why can `print_matrix` not be reused?
- Second representation (lecture): a one-dimensional array of `rows × cols` elements, where element (i, j) is at index $i \times cols + j$. Write `void print_flat(const int *m, int rows, int cols)` and check it on the same values stored in `int f[12]`. Which representation lets the same function print matrices of any shape? Verify that `print_flat(&m[0][0], 3, 4)` also works, and explain why (row-major order).

Q4. C strings. A C string is an array of `char` ending with the null byte `'\0'`.

- Write `size_t my_strlen(const char *s)`, a loop that stops at the null byte, and check it against `strlen` (`#include <string.h>`) on a few strings, including `""`. What is its complexity?
- Write a function `reverse_string(char *s)` that reverses a string in place (no return value), and a function `count_char(const char *s, char c)` that returns the number of occurrences of `c`. Test them on `char s[] = "Hello, world!"`.
- Predict, then check, what is printed by

```
char s[] = "Hello";
const char *t = "Hello";
printf("%zu %zu %zu %d %d\n", sizeof(s), sizeof(t), strlen(t), s == t, strcmp(s, t) == 0);
```

Where are `s` and `t` stored (which region of memory)? What does `s[0] == 'J'` do, and `t[0] == 'J'`?

Part B: Dynamic allocation

Q5. Meet the memory checker. Type in the following program, which contains the classic memory errors of the lecture, one per value of bug.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int bug; scanf("%d", &bug);
    int *p = malloc(4 * sizeof(int));
    for (int i = 0; i < 4; i++) p[i] = i;
    if (bug == 1) {
        p[4] = 4;
    } else if (bug == 2) {
        int a[4] = {0, 1, 2, 3}, i = 4;
        printf("%d\n", a[i]);
    }
```

```
} else if (bug == 3) {
    free(p);
    printf("%d\n", p[0]);
    return 0;
} else if (bug == 4) {
    free(p);
} else if (bug == 5) {
    p = malloc(8 * sizeof(int));
}
free(p);
return 0;
}
```

- Before running it, name the error made for each value of `bug` from 1 to 5.
- Compile it as usual and run it five times. Which errors have a visible effect? Why is this the worst possible behavior for a bug?

- (c) If the checker is available to you (see the reminders), compile it again with the sanitizer flags and run it five times. For each run, find in the report the kind of error (e.g., `heap-buffer-overflow`) and the line of your program where it happens, and the line where the block involved was allocated or freed.

Q6. An array of n values. Write a program that reads an integer n , then n integers, and prints their maximum and their mean (a `double`).

- Store the values in an array allocated with `malloc`, reuse `max_array` from Q2, and `free` the array at the end. Why can the array not simply be declared as `int a[n]`? (Try `int a[10000000]`; in `main` with $n = 10^7$: what happens, and what does the lecture call it? And if the same array is declared as a global variable?)
- Move the reading into a function `int *read_array(int *n)` that reads n into `*n`, allocates the array, fills it, and returns it, so that `main` becomes three lines. Write above the function a comment saying who owns the array, i.e., who must free it. First try to do it with a local array `int a[100]` returned by the function: what does `gcc` say?
- Remove the `free` and run the program: nothing visible happens, the memory is simply lost until the program exits. With the checker, if you have it, read the report, then put the `free` back and check that the report disappears.

Q7. Strings on the heap.

- Write `char *concat(const char *s, const char *t)`, returning a newly allocated string containing `s` followed by `t`, using `strlen` to compute the size to allocate and loops to copy the characters. Who frees the result? Forget the `+ 1` in the size: does the program still seem to work? It usually does, which is the danger; the checker, if you have it, reports the write of the null byte past the block.
- What is the complexity of the loop `for (size_t i = 0; i < strlen(s); i++)` over a string of length n ? Allocate a string of n characters `'a'` (do not forget the null byte) and count its `'a'`'s with two loops: one with this condition, one that computes the length once. Time both for $n = 50\,000$, $100\,000$, $200\,000$, $400\,000$ (without the checker!), apply the doubling test to the first loop, and compare with the second. Then compile with `-O2` and time again: what did the compiler do? (Is it allowed to, given that the loop does not modify `s`?)

Q8. What about Python? In Python, every value is an object on the heap, released automatically.

- `import sys`, then print `sys.getsizeof(x)` for `x` equal to `0`, `1`, `2**30`, `2**60`, `2**100`. Explain the values with the layout of a Python integer given in the lecture.
- Same for `[]` and `list(range(1000))`. The latter does not count the integer objects themselves: estimate the total memory of the list and compare with a C array of `1000 ints`. Compare with `sys.getsizeof(array.array("i", range(1000)))` (`import array`), which stores its values like C.

Part C: Collections in C++

Q9. Vectors. Rewrite Q6 in C++ with a `std::vector<int>` (`#include <vector>`): read n and then n integers with `std::cin`, appended with `push_back`, then print the maximum and the mean.

- Write the following functions, the first two with range-based `for` loops, the third reversing the vector in place:

```
int max_vector(const std::vector<int> &v);
double mean(const std::vector<int> &v);
void reverse(std::vector<int> &v);
```

C++

Which parameters are references, which are `const`, and why? What would a call of `max_vector` cost if its parameter were a plain `std::vector<int>`?

- Compare with your C program: what became of n , of `malloc`, of `free`?
- After reading the values, print `v[n]` and then `v.at(n)`: what happens in each case? Which behavior is the C one, which is the Python one?
- Copy the vector with `std::vector<int> w = v;`, modify `w[0]`, and print `v[0]` and `w[0]`, then `v == w`. What would the same lines do with Python lists (tutorial 3, Q11)? And with C arrays?

Q10. Strings. `std::string` (`#include <string>`) manages its own memory.

- (a) Rewrite `concat` of Q7 with `std::string`, then write `repeat(s, k)`, returning the string `s` repeated `k` times, taking `s` by `const` reference and using a loop with `+=`. Where did `strlen`, `malloc`, and `free` go?
- (b) Predict, then check, what the following prints, and compare with C strings (Q4(c)) and Python strings.

```
std::string s = "Hello", t = "Hello";
std::cout << (s == t) << " " << s.size() << "\n";
s += ", world";
std::string u = s;
u[0] = 'J';
std::cout << s << " | " << u << " | " << s.size() << "\n";
std::cout << s.substr(7, 5) << " " << s.find("world") << " " << (s.find("xyz") ==
↪ std::string::npos) << "\n";
```

C++

Q11. Counting words, in C++. Reuse `alice.txt` from tutorial 3 (or any long text). Write a program `freq.cpp` that reads words with `while (std::cin >> w)` and counts them in an `std::map<std::string, int>` (`#include <map>`); run it on the file as explained in the reminders.

- (a) Print the total number of words, the number of distinct words, and all the words appearing at least 200 times, with their counts. In which order are they printed? Compare with tutorial 3, Q14.
- (b) Words come with punctuation and capitals. Clean each word before counting it, with the following function (a character is an integer, lecture 1, and letters are consecutive):

```
std::string clean(const std::string &w) { // keep letters only, in lower case
    std::string r;
    for (char c : w) {
        if ('a' <= c && c <= 'z') r += c;
        if ('A' <= c && c <= 'Z') r += c - 'A' + 'a';
    }
    return r;
}
```

C++

Skip the words that become empty. How do the numbers change?

- (c) Find the most frequent word with a loop `for (const auto &[w, n] : freq)`. Print the number of words appearing exactly once.
- (d) Replace `std::map` with `std::unordered_map` (`#include <unordered_map>`): what changes in the output, and what does the lecture say about the cost of each operation?

Q12. Watching the capacity. Type in the lecture's C++ program printing `v.size()` and `v.capacity()` after each `push_back`, but only when the capacity changes.

- (a) Run it for 100 insertions. How many reallocations for n insertions? Add `v.reserve(100);` before the loop: what changes?
- (b) Time n calls to `v.push_back(i)` for $n = 2 \times 10^6, 4 \times 10^6, 8 \times 10^6, 1.6 \times 10^7$, then n calls to `v.insert(v.begin(), i)` for $n = 10^4, 2 \times 10^4, 4 \times 10^4, 8 \times 10^4$. Apply the doubling test and compare with the table of the lecture.

Part D: Dynamic arrays and amortized complexity

Q13. A dynamic array in C. Implement the lecture's `push_back` function, with the same four parameters, but following the pseudocode rather than using `realloc`: when the array is full, allocate a new array of twice the capacity (or 1), copy the elements one by one, free the old array, and update `*t` and `*cap`. Keep a global counter `long long copies` of the number of elements copied (one of the rare legitimate uses of a global variable: instrumenting a program to measure it).

- (a) Starting from `int *t = nullptr; int n = 0, cap = 0;`, append the integers from 0 to 99, printing `n` and `cap` whenever the capacity changes; compare with the lecture's output for `std::vector`. Free the array at the end (and check with the memory checker, if you have it).
- (b) For $n = 2 \times 10^6, 4 \times 10^6, 8 \times 10^6, 1.6 \times 10^7$ insertions, print `copies`, the ratio `copies/n`, and the time (without the checker). Apply the doubling test. Between which two bounds does `copies/n` stay, and why? What is the amortized cost of an insertion?

- (c) Replace the doubling with a growth by a constant: `*cap + 1000`. Same measurements for $n = 250\,000$, $500\,000$, 10^6 , 2×10^6 . What does the doubling test say now? Find a simple formula approximating `copies` as a function of n (you will prove it in Q14).

Q14. Amortized analysis, on paper. We insert n elements into an initially empty dynamic array; insertion i costs 1 if there is room, and i (copying $i - 1$ elements, writing one) if the array of capacity $i - 1$ is full and must be reallocated.

- Tripling.* The capacity goes $0, 1, 3, 9, 27, \dots$. For which values of i is insertion i expensive? How many expensive insertions among the first n ? Show with the direct method that the total cost is at most $n + 3n/2$, hence an amortized cost of $O(1)$ per insertion. (Recall $\sum_{k=0}^m 3^k = (3^{m+1} - 1)/2$.)
- Constant growth.* The capacity grows by a constant c each time: $0, c, 2c, 3c, \dots$. Compute the total cost of n insertions (with n a multiple of c), and the amortized cost of an insertion in $\Theta()$ notation. Compare with your measurements of Q13(c) for $c = 1000$.
- Shrinking, the wrong way.* Suppose that removing the last element halves the capacity as soon as the array is half full (size $\leq \text{cap}/2$), copying the elements into a new array. Exhibit a sequence of n operations on an array of m elements whose total cost is $\Theta(nm)$. Why does the lecture's rule (halve when a quarter full) avoid this?

Part E: If you have time

Q15. Potential method for tripling. Take $\Phi(T) = \alpha(3n(T) - \text{cap}(T))/2$ for an array of size $n(T)$ and capacity $\text{cap}(T)$, in the setting of Q14(a). Check that $\Phi \geq 0$ always holds and that $\Phi(T_0) = 0$; compute the amortized cost $\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$ of a cheap insertion and of an expensive one (with $c_i \leq \alpha$ and $c_i \leq \alpha i$ respectively), and conclude.

Q16. A binary counter. A counter from 0 to $2^k - 1$ is stored as an array of k bits, least significant first. Incrementing it flips the trailing 1s to 0 and the first 0 to 1.

- Implement `increment` (in Python or in C), counting the number of bit flips, and perform $n = 1000$ increments from 0. What is the worst-case cost of one increment? What is the total number of flips for $n = 1000, 10^4, 10^5, 10^6$, divided by n ?
- Explain the result with the potential method: take Φ equal to the number of 1 bits, and bound the amortized cost of an increment.

Q17. Shrinking, the right way. Add to Q13 a function `int pop_back(int **t, int *n, int *cap)` removing and returning the last element, and halving the capacity (copying into a new array) when the size becomes $\text{cap}/4$. Count copies for 10^6 insertions followed by 10^6 removals, then for 1000 repetitions of the alternating pair of operations of Q14(c) (start from an array with $\text{cap}/2 + 1$ elements). Try the “half full” rule instead and compare.

Check

Whether or not you reach the end of the sheet, by the end of the session you should be comfortable with: passing an array and its size to a C function, and knowing when it may modify it; having read at least one memory checker's report; allocating an array of n elements with `malloc` and freeing it exactly once, knowing who owns it; using `std::vector`, `std::string`, and `std::map` for what you would do with a Python list, string, or dictionary, passing them by `const` reference; explaining why doubling the capacity of a dynamic array gives $O(1)$ amortized insertion when a constant growth does not. If one of them is still unclear, ask us, or finish the corresponding exercises at home.