



UNIVERSITÉ
PARIS SCIENCES
& LETTRES

Memory Management, C/C++ Collections, & Amortized Complexity

The Art of Computer Programming

Pierre Senellart



5 October 2026

Outline

- + **Arrays in C**
- + **Memory management**
- + **Collections in C++**
- + **Amortized complexity**
- + **Dynamic arrays**
- + **Coda**

Arrays in C



Collections in C

- + Lecture 3: the C standard library provides **very limited support** for collections, essentially **fixed-size arrays**; no dynamic arrays, sets, or associative arrays
- + Any more advanced collection is either **written by the programmer** (with the tools of this lecture, and of the lecture on data structures) or taken from a **third-party library**
- + But arrays are the **building block** of all other data structures, and using them well requires understanding how the **memory** of a program is organized

C arrays

C

```
int a[10];           // 10 ints, uninitialized: garbage!
double d[3] = {1.0, 2.0, 3.0}; // 3 doubles, initialized
int b[] = {0, 1, 2}; // size 3, from the initializer
int z[100] = {1, 2}; // then 98 zeros: missing values are 0
a[0] = 42;           // indices go from 0 to 9
```

- + **Contiguous area of memory**: the elements are stored one after the other, so the address of `a[i]` is computed from that of `a[0]` in $O(1)$: **random access**
- + **Fixed size**, a constant known when compiling: no way to add or remove elements, and no built-in length, the programmer **keeps track of the size** (lecture 3: `binary_search(int t[], int n, int x)`)
- + **Mutable**, unless declared `const` (see further)
- + **No bounds checking**: `a[10]` or `a[-1]` is **undefined behavior** (crash, silent access to another variable, or anything else), where Python raises an `IndexError`

An array in memory

C

```
int a[4] = {5, 8, 13, 21};
```

	a[0]	a[1]	a[2]	a[3]	
4 bytes each (sizeof(int))	5	8	13	21	addresses (in hexadecimal)
	0x7ffc4a10	0x7ffc4a14	0x7ffc4a18	0x7ffc4a1c	

- ★ Lecture 1: &a[2] is the **address** of a[2], of type `int*` (**pointer** to an `int`); *p is the value at address p
- ★ Used as a value, the name of the array is the address of its first element: a is &a[0] (the array **decays** to a pointer)
- ★ **Pointer arithmetic**: a + i is a plus i × sizeof(int) bytes, the address of a[i]; a[i] is exactly *(a + i)
- ★ Here, a + 2 is 0x7ffc4a18 and *(a + 2) is 13; the same works with any pointer: after `int *p = a;`, p[3] is 21 and p++ makes p point to a[1]

Arrays and functions

- ★ When an array is passed to a function, only the **address of its first element** is passed: `void f(int a[])` is the same as `void f(int *a)` (lecture 3's `binary_search`)
- ★ Consequences: the function **does not know the size** of the array (pass it as an extra parameter); it can **modify** the elements, and the caller sees the modifications (pass by address, lecture 2); passing costs $O(1)$

C

```
int sum_array(const int *a, int n) {    // const: promise not to modify a
    int s = 0;
    for (int i = 0; i < n; i++) s += a[i];
    return s;
}

int main(void) {
    int a[] = {42, 26, 78, 59};
    printf("Sum: %d\n", sum_array(a, 4)); // Sum: 205
}
```

The sizeof operator

C

```
int a[10];  
printf("%zu\n", sizeof(int));           // 4 on most platforms  
printf("%zu\n", sizeof(a[0]));          // 4: one element  
printf("%zu\n", sizeof(a));             // 40: the whole array  
printf("%zu\n", sizeof(a) / sizeof(a[0])); // 10: the length
```

- + `sizeof` gives the size **in bytes** of a type, a variable, or an expression, as a value of the unsigned type `size_t` (printed with `%zu`); it is computed **when compiling**
- + Applied to an array variable, it gives the size of the **whole array**: the usual way to get the length of an array **in the scope where it was declared**
- + **Warning**: inside a function with an `int *a` or `int a[]` parameter, `sizeof(a)` is the size of a **pointer** (8 bytes on 64-bit platforms), not of the array: the length information is lost, hence the extra parameter

Common operations on C arrays

Type `T a[N];` array of N elements of type T

Initialization `int a[3] = {0, 1, 2};` or `int a[] = {0, 1, 2};`

Access `a[i]`

$O(1)$

Modification `a[i] = 42;`

$O(1)$

Length `sizeof(a) / sizeof(a[0])` (only where `a` was declared)

$O(1)$

Loop `for (int i = 0; i < n; i++) { /* use a[i] */ }`

$O(n)$

Membership test a loop, written by the programmer

$O(n)$

Copy `for (int i = 0; i < n; i++) b[i] = a[i];` or
`memcpy(b, a, sizeof(a))`

$O(n)$

Insert, remove impossible: fixed size; build a new array

$O(n)$

Traps: `b = a;` is not allowed (arrays are not assignable); `a == b` compares the addresses, not the contents; `memcpy` and `memcmp` are in `<string.h>`

Two-dimensional arrays

C

```
int m[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};

int trace(int m[][4], int rows) { // column count: part of the type
    int t = 0;
    for (int i = 0; i < rows; i++) t += m[i][i];
    return t;
}
```

- + An array of arrays (here, 3 rows of 4 columns), stored **row by row** in one contiguous block (**row-major** order): $m[i][j]$ is at position $4i + j$
- + When passed to a function, all dimensions **but the first** must be given, so that the compiler can compute $4i + j$
- + Dimensions known only at run time: one array of size $rows \times cols$ indexed by $i * cols + j$

Character strings are arrays of char

C

```
#include <string.h>
char s[] = "Hello";           // array of 6 chars: 'H' 'e' 'l' 'l' 'o' '\0'
const char *t = "Hello";     // pointer to a read-only string literal
printf("%zu %zu\n", sizeof(s), strlen(s)); // 6 5
s[0] = 'J';                   // now "Jello"; t[0] = 'J': undefined behavior
```

- ★ Lecture 1: a C string is a sequence of bytes ending with the **null byte** `'\0'`; the length is not stored
- ★ `strlen` scans the bytes until the null byte: **$O(n)$** , where Python's `len(s)` is $O(1)$; avoid `for (i = 0; i < strlen(s); i++)`: $O(n^2)$!
- ★ **No bounds checking**: `strcpy(buf, "Hello, world!")` into `char buf[8]` is the classic **buffer overflow**, historically the most common security vulnerability
- ★ Prefer the functions taking a maximal size (`snprintf`, `strncpy`), or `std::string` in C++ (later in this lecture)

Constants

- + `const` declares data as **read-only**: modifying it is a compilation error

```
const int a = 42;           // a cannot be modified
const int b[3] = {0, 1, 2}; // immutable array
const double PI = 3.141592653589793; // constants: often uppercase
```

C

- + Also applies to what a **pointer points to**, or to the pointer itself (read right to left):
 - `const int *p`; pointer to a constant integer: `*p = 1` forbidden, `p = q` allowed
 - `int * const p`; constant pointer to an integer: `p = q` forbidden, `*p = 1` allowed
 - `const int * const p`; constant pointer to a constant integer
- + **Best practice**: declare a function parameter `const T *a` whenever the function does not modify the array (as in `sum_array`): the compiler **checks** the promise, and readers of the code know an array passed to this function is left intact

Memory management



So far: local variables on the call stack

- ★ Lecture 2: local variables and parameters live on the **call stack**, reserved when a function is called, freed when it returns; arrays declared in a function too:
`int t[1000]`; takes 4 000 bytes of stack
- ★ The stack is **small** (typically 8 MB on Linux, 1 MB on Windows): `int t[10000000]`; in a function crashes the program (**stack overflow**)
- ★ Three needs the stack cannot fulfill:
 1. data whose **size** is only known when the program runs (e.g., an array of n values, n read from the input)
 2. data that must **outlive** the function that creates it
 3. data **shared** by several functions, without passing it around as a parameter

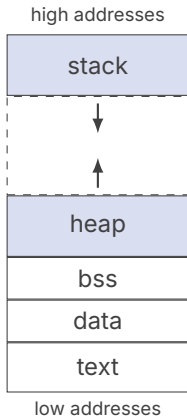
Returning a local array: wrong

C

```
int *make_array(void) {  
    int t[3] = {1, 2, 3};  
    return t;           // address of a variable that no longer exists  
}  
  
int main(void) {  
    int *p = make_array();  
    printf("%d\n", p[0]); // undefined behavior: p is a dangling pointer  
}
```

- ✦ The array `t` is freed when `make_array` returns, and its location reused by the next calls: `p[0]` may hold anything, writing to it corrupts other data
- ✦ `gcc -Wall` warns: function returns address of local variable; the fix is dynamic allocation (see further)

Memory of a process



The memory of a running program (a **process**) is made of several **regions** (or **segments**):

stack local variables, parameters, return addresses

heap **dynamically allocated** data (see further); grows on request

bss uninitialized **global** variables

data initialized **global** variables, string literals, other constants

text the program itself, as machine code (lecture 1)

Addresses are **virtual**: the operating system maps them to the physical RAM, each process seeing its own isolated memory

Global variables

- Variables **accessible from all functions**, with one fixed address and an **extent** (lecture 2) equal to the whole duration of the program
- Either **initialized** at declaration, stored in **data**, or **uninitialized**, stored in **bss** and set to a default value when the program starts (0 for numbers in C/C++); neither on the stack nor on the heap
- Common and useful for **constants** (conventionally in uppercase)
- For anything else, **use only when really needed**: a function that reads or writes a global variable cannot be understood in isolation, is hard to test, and behaves incorrectly when the program runs the same function several times in parallel

Global variables in C/C++

- Declared **outside of any function**, with the usual syntax; usable by all functions that **follow** the declaration in the file, unless shadowed by a local variable of the same name
- Initialized once, before `main` starts; a global variable without initializer starts at 0

```
const double PI = 3.141592653589793; // constant (data segment)
int calls;                             // variable (bss), starts at 0

double perimeter(double radius) {
    calls++;                            // readable and writable anywhere
    return 2 * PI * radius;
}
```

C

Static variables in C/C++

- ★ A local variable declared `static` is a global variable in disguise: stored in data or bss, initialized once, it **keeps its value** from one call to the next; only its **scope** is limited to the block (**use sparingly!**)

C

```
void counter(void) {  
    static int count = 0; // initialized once, not at each call  
    count++;  
    printf("Count = %d\n", count);  
}  
  
int main(void) {  
    counter(); // prints Count = 1  
    counter(); // prints Count = 2  
}
```

Global variables in Python

- Variables assigned **outside of any function** are global to the file (**module**, lecture 6)
- A function can **read** a global variable by naming it; but an **assignment** inside a function creates a **local** variable (lecture 2), unless the variable is declared **global** in the function

Python

```
PI = 3.141592653589793
count = 0
def perimeter(radius):
    return 2 * PI * radius # reading a global variable: nothing special
def increment():
    global count           # without this line, count += 1 is an error
    count += 1
increment()
print(perimeter(1), count) # 6.283185307179586 1
```

Dynamic memory allocation

- ★ So far: local variables (stack), global and static variables (data, bss): fixed size and lifetime, determined by the **structure of the program**

Dynamic memory allocation

- ★ So far: local variables (stack), global and static variables (data, bss): fixed size and lifetime, determined by the **structure of the program**
- ★ Variables whose **size or lifetime** is only decided **at run time** are stored in a third region, the **heap**

Dynamic memory allocation

- ★ So far: local variables (stack), global and static variables (data, bss): fixed size and lifetime, determined by the **structure of the program**
- ★ Variables whose **size or lifetime** is only decided **at run time** are stored in a third region, the **heap**
- ★ They are created (**allocated**) and destroyed (**released, freed**) explicitly by the program, or automatically on its behalf

Dynamic memory allocation

- ★ So far: local variables (stack), global and static variables (data, bss): fixed size and lifetime, determined by the **structure of the program**
- ★ Variables whose **size or lifetime** is only decided **at run time** are stored in a third region, the **heap**
- ★ They are created (**allocated**) and destroyed (**released, freed**) explicitly by the program, or automatically on its behalf
- ★ Typical uses:

Dynamic memory allocation

- + So far: local variables (stack), global and static variables (data, bss): fixed size and lifetime, determined by the **structure of the program**
- + Variables whose **size or lifetime** is only decided **at run time** are stored in a third region, the **heap**
- + They are created (**allocated**) and destroyed (**released, freed**) explicitly by the program, or automatically on its behalf
- + Typical uses:
 - **variable-size** data: arrays of n elements, dynamic arrays, sets, associative arrays, strings...

Dynamic memory allocation

- + So far: local variables (stack), global and static variables (data, bss): fixed size and lifetime, determined by the **structure of the program**
- + Variables whose **size or lifetime** is only decided **at run time** are stored in a third region, the **heap**
- + They are created (**allocated**) and destroyed (**released, freed**) explicitly by the program, or automatically on its behalf
- + Typical uses:
 - **variable-size** data: arrays of n elements, dynamic arrays, sets, associative arrays, strings...
 - values that **outlive** the function that created them

Dynamic memory allocation

- + So far: local variables (stack), global and static variables (data, bss): fixed size and lifetime, determined by the **structure of the program**
- + Variables whose **size or lifetime** is only decided **at run time** are stored in a third region, the **heap**
- + They are created (**allocated**) and destroyed (**released, freed**) explicitly by the program, or automatically on its behalf
- + Typical uses:
 - **variable-size** data: arrays of n elements, dynamic arrays, sets, associative arrays, strings...
 - values that **outlive** the function that created them
- + **Cost**: every allocation must be matched by a release, once the value is not needed any more; getting this right is the **main difficulty** of programming in C

Explicit vs. automatic memory management

- + **Memory management**: deciding when heap memory is allocated and released; two main approaches:
 - Explicit management** the programmer writes the allocations and the releases (C, C++)
 - Automatic management** the run-time system detects the values that the program can no longer reach and reclaims their memory: **garbage collection** (Python, Java, JavaScript, OCaml...)
- + Explicit management gives **fine control** over memory and time, at the risk of **memory leaks** (memory never released), **dangling pointers** (memory used after release), **double releases**
- + Garbage collection gives **ease of use** and **safety**, at the price of some overhead and of less predictable pauses

Allocation and release in C

C

```
int *t = malloc(n * sizeof(int));           // n ints, uninitialized
double *z = calloc(n, sizeof(double));      // n doubles, all set to 0
if (t == nullptr) { /* not enough memory: handle the failure */ }
t[0] = 42;                                  // used exactly like an array
t = realloc(t, 2 * n * sizeof(int));        // now 2n ints, the first n kept
free(t); free(z);                          // release: t, z no longer valid
```

- + `malloc` (in `<stdlib.h>`) returns the **address** of a block of the requested number of **bytes** on the heap, or a **null pointer** (see further) if impossible; its type `void*` converts to any pointer type
- + The block **does not know its size**: keep `n` alongside `t`; `realloc` may **move** the block (copying it): use the returned pointer
- + Every block must be released **exactly once** with `free`, and never used afterwards

Allocation and release in C++

C++

```
double *pd = new double(3.14); // 1 double, initialized to 3.14
double *ad = new double[n];    // n doubles, uninitialized
int *az = new int[n]();        // n ints, all set to 0
delete pd;                     // release what was allocated by new
delete[] ad; delete[] az;      // release what was allocated by new[]
```

- + `new` and `new[]` know the type of what they allocate (no `sizeof`) and can **initialize** it; on failure they throw an **exception** (lecture 8) instead of returning a null pointer
- + `delete` and `delete[]` must **match** the allocation; never mix them with `malloc/free`
- + In modern C++, `new` and `delete` are **rarely written by hand**: the collections of the standard library (next section) manage their memory themselves; in this course, we use them only in the **implementation** of data structures

Returning an array from a function, correctly

C

```
int *squares(int n) {                // the n first squares
    int *t = malloc(n * sizeof(int));
    if (t == nullptr) return nullptr;
    for (int i = 0; i < n; i++) t[i] = i * i;
    return t;                        // the block outlives the function
}

int main(void) {
    int *s = squares(10);
    if (s == nullptr) return 1;
    printf("%d\n", s[7]);             // 49
    free(s);                         // the caller must release it
    return 0;
}
```

Convention to document: who **owns** a block, i.e., is responsible for freeing it (here, the caller)

Null pointers

- ✦ `nullptr` (C23 and C++) is the **null pointer**: a pointer that **points to nothing**; assignable to any pointer type to mean “no value” (allocation failure, element not found, end of a list...)
- ✦ Older C code and the documentation spell it `NULL`, a macro (in C, the address 0) that still works; prefer `nullptr`, which cannot be confused with the integer 0
- ✦ In a test, a pointer is **true** iff it is not null
- ✦ Dereferencing a null pointer is undefined behavior, in practice an immediate crash (**segmentation fault**): the best kind of memory bug, deterministic and easy to locate

```
int *p = nullptr;
/* ... */
if (p) *p = 42; // only if p is not null
```

C

The classic memory errors

memory leak

```
// initial allocation
int *p = malloc(4 * sizeof(int));
p = malloc(8 * sizeof(int));
// the first block can never be freed
```

use after free, double free

```
free(p);
p[0] = 1; // dangling pointer
free(p); // double free
```

out-of-bounds access

```
// allocating 4 ints
int *q = malloc(4 * sizeof(int));
q[4] = 1; // writes past the block
```

uninitialized pointer

```
int *r;
*r = 1; // r holds a garbage address
```

- + All are **undefined behavior**: crash, silent data corruption, security vulnerability, or a program that works on one machine only
- + **Best practices**: initialize every pointer, decide who owns each block, and **test with a memory checker** (gcc -g -fsanitize=address,undefined, valgrind), which reports the faulty line

What about Python?

- Python uses **automatic memory management**: every value (even an integer) is an **object on the heap**; a variable holds the **address** of its object (lecture 1: `id(x)`), and the interpreter calls `malloc` and `free` on your behalf
- CPython counts the **references** to each object (variables, list elements...) and frees it as soon as the count drops to zero; a **garbage collector** handles objects referencing each other in a cycle
- Easier and safer, but with a cost: a Python `int` occupies 28 bytes instead of 4, and every access follows a pointer (part of the constant factor of lecture 3)

Python

```
a = [1, 2, 3] # a list object on the heap, a holds its address
b = a        # same list, now with 2 references
del a        # 1 reference left
b = None     # 0 references: the list is freed immediately
```

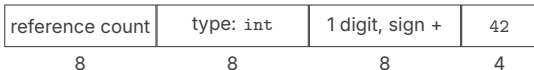
Why does a Python integer take 28 bytes?

C: `int x = 42;`



4 bytes

Python: `x = 42`



28 bytes

- ✦ Every Python value is an **object** on the heap with the same **header**: a reference count (for automatic release) and a pointer to its type (**dynamic typing**, lecture 1)
- ✦ Python integers have **arbitrary size** (lecture 1: no overflow): 30-bit **digits**, whose number and sign are stored too; 28 bytes up to 2^{30} , 32 up to 2^{60} , etc.
- ✦ In a list, add the 8-byte pointer: 36 bytes per element against 4 for a C `int`, plus a memory access to follow it
- ✦ CPython **shares** the integers from -5 to 256 ; floats (24 bytes) and strings have the same header

Collections in C++



Collections in the C++ standard library

- + C++ provides in its standard library the collections (**containers**) that C lacks: dynamic arrays, strings, sets, associative arrays... each in its own header:
`#include <vector>, #include <string>, #include <map>...`
- + They are **templates (generic types**, lecture 12): the type of the elements is given between angle brackets, and all elements have this type (**type-safe**):
`std::vector<int>, std::vector<std::string>, std::set<double>,
std::map<std::string, int>`
- + They manage their own heap memory: allocated as the collection grows, **released automatically** when the variable goes out of scope, no `new` or `delete` to write
- + **Efficient**: a `std::vector<int>` stores its integers contiguously, like a C array, not through pointers to objects as a Python list does

Namespaces

- Names of the C++ standard library are grouped in the **namespace** `std`, to avoid clashes with the programmer's or third-party names: `std::vector`, `std::cout`...
- Alternatives to writing `std::` everywhere:

C++

```
#include <iostream>
#include <vector>
using std::vector;           // bring one name into scope
vector<int> v;
using namespace std;        // bring all names of std into scope
vector<int> v2;
cout << "Hi!" << endl;     // std::cout, std::endl
```

- `using namespace std`; is fine in small programs (tutorials); avoided in large programs, and never in header files (lecture 6)

Iterating over collections

C++

```
std::vector<int> v = {1, 2, 3};  
for (int x : v) std::cout << x << "\n"; // x: a copy of each element  
for (int &x : v) x *= 2; // reference: modifies in place  
for (const auto &x : v) std::cout << x; // auto: inferred type; no copy  
for (size_t i = 0; i < v.size(); i++) // loop over indices  
    std::cout << v[i] << "\n";
```

- ★ The **range-based for** works on every container, like Python's; **break** and **continue** as usual
- ★ **auto** lets the compiler **infer the type** of a variable from its initializer; convenient for long type names, as in **const auto** &x to loop without copying, or **auto** [key, value] to unpack a pair (see further)
- ★ Sizes have the unsigned type **size_t**: comparing them with an **int** triggers a warning with **-Wall -Wextra**

C++ containers

Name (in std)	Abstract type	Mutable?	Size
<code>array<T, N></code>	Random-access ordered sequence	Yes	Fixed
<code>vector<T></code>	Random-access ordered sequence	Yes	Dynamic
<code>string</code>	Random-access ordered sequence (bytes)	Yes	Dynamic
<code>deque<T></code>	Random-access ordered sequence	Yes	Dynamic
<code>list<T></code>	Ordered sequence, no random access	Yes	Dynamic
<code>set<T></code>	Value-based collection, sorted	Yes	Dynamic
<code>unordered_set<T></code>	Value-based collection	Yes	Dynamic
<code>map<K, V></code>	Associative array, sorted by key	Yes	Dynamic
<code>unordered_map<K, V></code>	Associative array	Yes	Dynamic

- + Immutability: declare the variable `const`; `std::array<T, N>`: a C array that **knows its size** and can be copied
- + `deque`, `list`, and the data structures behind sets and maps: lecture 6

Vectors (like Python lists)

Type `std::vector<T>`, ordered sequence of variable size n

Construction `std::vector<int> v;` `std::vector<int> v = {1, 2, 3};`
`std::vector<int> v(n, 0);` (n zeros)

Access `v[i]` (unchecked) `v.at(i)` (exception if out of bounds) **$O(1)$**

Modification `v[i] = 42;` **$O(1)$**

Append at end `v.push_back(42);` **$O(1)$ (amortized)**

Insert elsewhere `v.insert(v.begin() + i, 42);` **$O(n)$**

Remove at end `v.pop_back();` **$O(1)$**

Remove elsewhere `v.erase(v.begin() + i);` **$O(n)$**

Length `v.size()` **$O(1)$**

Membership test `std::find(v.begin(), v.end(), x) != v.end()` (`<algorithm>`) **$O(n)$**

Unlike C arrays, vectors can be **assigned** and **compared** (`w = v`; `v == w`, element by element, $O(n)$); `v.begin() + i` is the position of the i -th element (an **iterator**, lecture 12)

Strings (like Python str)

Type	<code>std::string</code>	<code>#include <string></code>
Definition	ordered sequence of bytes of variable size n , mutable	
Construction	<code>std::string s; std::string s = "Hello"; std::to_string(42)</code>	
Access	<code>s[i]</code>	$O(1)$
Modification	<code>s[i] = 'a';</code>	$O(1)$
Append	<code>s += "abc"; s += c;</code>	$O(1)$ (amortized) per byte
Concatenation	<code>s + t</code> (new string)	$O(n_1 + n_2)$
Comparison	<code>s == t s < t</code> (lexicographic)	$O(\min(n_1, n_2))$
Length	<code>s.size()</code>	$O(1)$
Substring	<code>s.substr(i, len)</code>	$O(len)$
Search	<code>s.find("lo")</code> (position, or <code>std::string::npos</code>)	$O(n)$

Unlike C strings: length stored (no null byte to scan), assignment and comparison operators work, memory grows as needed; `s.c_str()` gives a `const char*` for C functions, `std::stoi(s)` converts to an integer

Sets (like Python set)

Type `std::set<T>` (sorted) `std::unordered_set<T>` (hash table)

Definition value-based collection of variable size n , mutable

Construction `std::set<int> s;` `std::set<int> s = {1, 2, 3};`

Membership test `s.count(x)` `s.find(x) != s.end()` $O(\log n)$ / $O(1)$ on average

Add `s.insert(x);` $O(\log n)$ / $O(1)$ on average

Remove `s.erase(x);` $O(\log n)$ / $O(1)$ on average

Length `s.size()` $O(1)$

Loop `for (const auto &x : s)` (in increasing order for `std::set`) $O(n)$

- Complexities: `std::set` / `std::unordered_set`; the latter is like Python's `set`: $O(1)$ on average, $O(n)$ in the worst case (lecture 6); the former keeps its elements **sorted**, which `set` does not
- Elements must be comparable with `<` (`set`) or hashable (`unordered_set`); true for numbers and strings

Maps (like Python dict)

Type `std::map<K, V>` (sorted by key) `std::unordered_map<K, V>` (hash table)

Definition associative array of variable size n from keys of type K to values of type V

Construction `std::map<std::string, int> m;`
`std::map<std::string, int> m = {{"a", 1}, {"b", 2}};`

Membership test `m.count(key)` `m.find(key) != m.end()` $O(\log n)$ / $O(1)$ on average

Access `m[key]` `m.at(key)` (exception if absent) $O(\log n)$ / $O(1)$ on average

Insert/Update `m[key] = value;` $O(\log n)$ / $O(1)$ on average

Remove `m.erase(key);` $O(\log n)$ / $O(1)$ on average

Length `m.size()` $O(1)$

Loop `for (const auto &[key, value] : m)` $O(n)$

Trap: `m[key]` on an absent key **inserts** it with a default value (0, empty string...);
handy for counting (`m[word]++`), but not a membership test: use `count` or `find`

Example: word frequencies

C++

```
#include <iostream>
#include <map>
#include <string>
#include <vector>

int main() {
    std::vector<std::string> words =
        {"to", "be", "or", "not", "to", "be"};
    std::map<std::string, int> freq;
    for (const auto &w : words) freq[w]++;
    for (const auto &[w, n] : freq)
        std::cout << w << ": " << n << "\n";
    return 0;
}
```

Python

```
freq = {}
words = ["to", "be", "or", "not",
         "to", "be"]
for w in words:
    freq[w] = freq.get(w, 0) + 1
for w in sorted(freq):
    print(f"{w}: {freq[w]}")
```

Output

```
be: 2
not: 1
or: 1
to: 2
```

`std::map` enumerates its keys in sorted order;
with `std::unordered_map`, the order would be
arbitrary

Passing collections to functions

- A container passed **by value** is **copied**: $O(n)$ time and memory per call, and modifications are lost
- Pass **by reference** instead (lecture 2), with **const** when the function only reads it:

C++

```
int sum(const std::vector<int> &v) { // no copy, no modification
    int s = 0;
    for (int x : v) s += x;
    return s;
}

void append_squares(std::vector<int> &v, int n) { // modifies v in place
    for (int i = 0; i < n; i++) v.push_back(i * i);
}
```

- **Rule of thumb**: **const** T & for any parameter larger than a few bytes that is only read; T & to modify it; plain T for numbers

Python and C++ collections side by side

Need	Python	C++
Sequence of variable size	<code>list</code>	<code>std::vector<T></code>
Sequence of fixed size	<code>tuple</code>	<code>std::array<T, N></code> , <code>std::pair<T1, T2></code>
Character string	<code>str</code>	<code>std::string</code>
Set	<code>set</code>	<code>std::unordered_set<T></code> (<code>std::set<T></code> : sorted)
Associative array	<code>dict</code>	<code>std::unordered_map<K, V></code> (<code>std::map<K, V></code> : sorted)

- ★ Same abstract kinds of collections (lecture 3), similar complexities: what you know about choosing a Python collection transfers to C++
- ★ Differences: C++ containers are **homogeneous** (one element type), copied on assignment, and `std::set`/`std::map` offer a **sorted** enumeration that Python's hash-based collections do not
- ★ Reference documentation: <https://en.cppreference.com/w/cpp/container>

Amortized complexity



Amortized complexity

- ★ Lecture 3: appending to a Python list is " $O(1)$ amortized"; so far, complexity was bounded **separately for each operation** of a data structure

Amortized complexity

- ✦ Lecture 3: appending to a Python list is " $O(1)$ amortized"; so far, complexity was bounded **separately for each operation** of a data structure
- ✦ Sometimes, no good bound holds for each individual operation (some are occasionally expensive), but a good bound holds for the **total time** of any **sequence** of operations

Amortized complexity

- ★ Lecture 3: appending to a Python list is “ $O(1)$ amortized”; so far, complexity was bounded **separately for each operation** of a data structure
- ★ Sometimes, no good bound holds for each individual operation (some are occasionally expensive), but a good bound holds for the **total time** of any **sequence** of operations
- ★ Let $o_1, \dots, o_n$ be a sequence of n operations on a data structure, with costs $c_1, \dots, c_n$; the **amortized complexity** of an operation is the average

$$\frac{1}{n} \sum_{i=1}^n c_i$$

for the **worst possible sequence** of n operations

Amortized complexity

- ★ Lecture 3: appending to a Python list is “ $O(1)$ amortized”; so far, complexity was bounded **separately for each operation** of a data structure
- ★ Sometimes, no good bound holds for each individual operation (some are occasionally expensive), but a good bound holds for the **total time** of any **sequence** of operations
- ★ Let $o_1, \dots, o_n$ be a sequence of n operations on a data structure, with costs $c_1, \dots, c_n$; the **amortized complexity** of an operation is the average

$$\frac{1}{n} \sum_{i=1}^n c_i$$

for the **worst possible sequence** of n operations

- ★ Only meaningful once the **kind of sequence** is specified (e.g., n insertions from an empty structure)

Amortized vs. average-case complexity

- + Two **orthogonal** notions, both averages, but not over the same thing:

Amortized vs. average-case complexity

- + Two **orthogonal** notions, both averages, but not over the same thing:
Amortized complexity average over the **operations of a sequence**, for the worst sequence: no probability involved, the bound on the total time is a **guarantee**

Amortized vs. average-case complexity

- + Two **orthogonal** notions, both averages, but not over the same thing:

Amortized complexity average over the **operations of a sequence**, for the worst sequence: no probability involved, the bound on the total time is a **guarantee**

Average-case complexity average over the **possible inputs** of one operation (lecture 3), as opposed to the worst case; an assumption on the distribution of inputs is needed

Amortized vs. average-case complexity

- + Two **orthogonal** notions, both averages, but not over the same thing:
 - Amortized complexity** average over the **operations of a sequence**, for the worst sequence: no probability involved, the bound on the total time is a **guarantee**
 - Average-case complexity** average over the **possible inputs** of one operation (lecture 3), as opposed to the worst case; an assumption on the distribution of inputs is needed
- + The usual notion is therefore **amortized worst-case** complexity; amortized average-case complexity also exists (e.g., insertion in a hash table, lecture 6)

Amortized vs. average-case complexity

- Two **orthogonal** notions, both averages, but not over the same thing:
 - Amortized complexity** average over the **operations of a sequence**, for the worst sequence: no probability involved, the bound on the total time is a **guarantee**
 - Average-case complexity** average over the **possible inputs** of one operation (lecture 3), as opposed to the worst case; an assumption on the distribution of inputs is needed
- The usual notion is therefore **amortized worst-case** complexity; amortized average-case complexity also exists (e.g., insertion in a hash table, lecture 6)
- Example: a dynamic array that is occasionally copied in full has $\Theta(n)$ worst-case insertion, $O(1)$ amortized insertion; linear search has $\Theta(n)$ worst-case and $\Theta(n)$ average-case complexity, and amortization is irrelevant since searches are independent

Computing an amortized complexity

- ★ **Directly (aggregate method)**: count the cost of each operation of the worst sequence, sum, and divide by n

Computing an amortized complexity

- + **Directly (aggregate method)**: count the cost of each operation of the worst sequence, sum, and divide by n
- + Sometimes **difficult**: the cost of an operation depends on the state left by all previous ones

Computing an amortized complexity

- + **Directly (aggregate method)**: count the cost of each operation of the worst sequence, sum, and divide by n
- + Sometimes **difficult**: the cost of an operation depends on the state left by all previous ones
- + **Trick (potential method)**: associate a **potential** $\Phi(X) \geq 0$ to each state X of the data structure, a reserve of prepaid work; a cheap operation **increases** the potential (it pays a little more than its cost), an expensive operation **decreases** it (it consumes what was prepaid)

Computing an amortized complexity

- + **Directly (aggregate method)**: count the cost of each operation of the worst sequence, sum, and divide by n
- + Sometimes **difficult**: the cost of an operation depends on the state left by all previous ones
- + **Trick (potential method)**: associate a **potential** $\Phi(X) \geq 0$ to each state X of the data structure, a reserve of prepaid work; a cheap operation **increases** the potential (it pays a little more than its cost), an expensive operation **decreases** it (it consumes what was prepaid)
- + Each operation can then be analyzed **individually**: its effect on the future is replaced by its effect on the potential

Computing an amortized complexity

- ✦ **Directly (aggregate method)**: count the cost of each operation of the worst sequence, sum, and divide by n
- ✦ Sometimes **difficult**: the cost of an operation depends on the state left by all previous ones
- ✦ **Trick (potential method)**: associate a **potential** $\Phi(X) \geq 0$ to each state X of the data structure, a reserve of prepaid work; a cheap operation **increases** the potential (it pays a little more than its cost), an expensive operation **decreases** it (it consumes what was prepaid)
- ✦ Each operation can then be analyzed **individually**: its effect on the future is replaced by its effect on the potential
- ✦ The difficulty moves to **finding the right potential function**

Potential method

- Let X_0 be the initial state, X_i the state after operation o_i (of real cost c_i), and Φ a potential function on states; the **amortized cost** of o_i is

$$\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$$

Potential method

- Let X_0 be the initial state, X_i the state after operation o_i (of real cost c_i), and Φ a potential function on states; the **amortized cost** of o_i is

$$\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$$

- The sum telescopes:

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n c_i + \Phi(X_n) - \Phi(X_0), \quad \text{so} \quad \sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i \quad \text{as soon as } \Phi(X_n) \geq \Phi(X_0)$$

(e.g., $\Phi \geq 0$ everywhere and $\Phi(X_0) = 0$ for the empty structure)

Potential method

- Let X_0 be the initial state, X_i the state after operation o_i (of real cost c_i), and Φ a potential function on states; the **amortized cost** of o_i is

$$\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$$

- The sum telescopes:

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n c_i + \Phi(X_n) - \Phi(X_0), \quad \text{so} \quad \sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i \quad \text{as soon as } \Phi(X_n) \geq \Phi(X_0)$$

(e.g., $\Phi \geq 0$ everywhere and $\Phi(X_0) = 0$ for the empty structure)

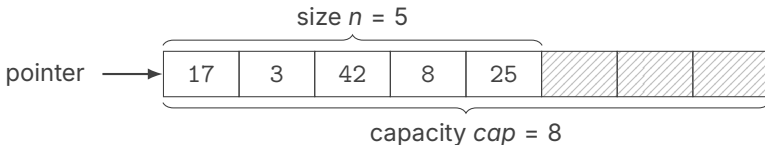
- Therefore, if every operation has amortized cost $\hat{c}_i \leq C$, the real total cost is at most nC : the amortized complexity of an operation is **$O(C)$** , and it suffices to bound $\hat{c}_i$ **for each operation separately**

Dynamic arrays



Dynamic array data structure

- ✦ The data structure behind Python lists, `std::vector`, `std::string`...
- ✦ A pointer to a fixed-size array (on the heap) of **capacity** cap , and an integer $n \leq cap$, the **size**: the number of cells actually used, always the first ones



- ✦ **Random access** in $\Theta(1)$; removal of the last element in $\Theta(1)$: decrement n
- ✦ Insertion at the end in $\Theta(1)$ if $n < cap$; otherwise, a **larger array** is allocated and the elements **copied**: $\Theta(n)$, but rare enough for an **amortized $\Theta(1)$** , as we now show
- ✦ Insertion or removal elsewhere in $\Theta(n)$: the following elements must be shifted

Insertion at the end of a dynamic array

Input: array T of capacity cap and size n ,
element x

Output: array T of size $n + 1$ with $T[n] = x$

if $n = cap$ then

$cap' \leftarrow \max(2 \times cap, 1)$

create a new array T' of capacity cap'

for $i \leftarrow 0$ to $n - 1$ do

$T'[i] \leftarrow T[i]$

end for

destroy the array T

$T \leftarrow T'$; $cap \leftarrow cap'$

end if

$T[n] \leftarrow x$

$n \leftarrow n + 1$

C

```
// Append x to (*t, *n, *cap)
void push_back(int **t, int *n, int *cap,
               int x) {
    if (*n == *cap) {
        *cap = *cap == 0 ? 1 : 2 * *cap;
        *t = realloc(*t, *cap * sizeof(int));
    }
    (*t)[*n] = x;
    (*n)++;
}
```

`realloc` does the allocation, copy, and release (a real program checks its result for `nullptr`); the three variables are passed **by address** (lecture 2) since the function modifies them; next lecture, a **struct** will group them into one value

Watching the capacity grow

C++

```
std::vector<int> v;  
for (int i = 0; i < 100; i++) {  
    v.push_back(i);  
    std::cout << v.size() << ":"  
               << v.capacity() << " ";  
}
```

Output

```
1:1 2:2 3:4 5:8 9:16  
17:32 33:64 65:128
```

Python

```
import sys  
l = []  
for i in range(100):  
    l.append(i)  
    size = sys.getsizeof(l) - sys.getsizeof([])  
    print(len(l), ":", size // 8) # 8-byte pointers
```

Output

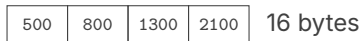
```
1:4 5:8 9:16 17:24 25:32 33:40  
41:52 53:64 65:76 77:92 93:108
```

- + Output shown only when the capacity changes; g++ 13: exact doubling; CPython 3.12: growth factor of about 1.125
- + Any factor > 1 works: the size after k reallocations is exponential in k , hence **$O(\log n)$ reallocations** for n insertions; growing the capacity by a **constant** would give $\Theta(n)$ amortized insertion (why?)

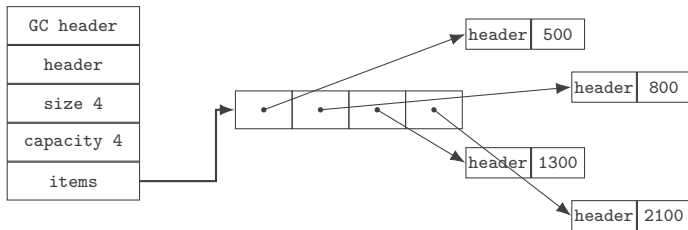
Dynamic arrays in Python

- Python **lists** are dynamic arrays... of **pointers** to objects, which is how they can be **heterogeneous**
- `array.array` (standard library) and NumPy arrays store **homogeneous** values contiguously, as in C: less memory, faster processing

C: `int t[4] = {500, 800, 1300, 2100};`



Python: `l = [500, 800, 1300, 2100]`



About 200 bytes: the list object (56, including the garbage collector's header), 4 pointers, 4 integer objects of 28 bytes each

Amortized complexity of n insertions: direct (1/2)

- Start from an empty array T_0 (capacity 0) and insert **n elements**, giving arrays $T_1, \dots, T_n$ of sizes $1, \dots, n$; let c_i be the cost of the i -th insertion

Amortized complexity of n insertions: direct (1/2)

- Start from an empty array T_0 (capacity 0) and insert **n elements**, giving arrays $T_1, \dots, T_n$ of sizes $1, \dots, n$; let c_i be the cost of the i -th insertion
- Insertion i is **expensive** when T_{i-1} is full: $i - 1 = \text{cap}(T_{i-1})$; the $i - 1$ elements are then copied, so **$c_i = \Theta(i)$** ; otherwise **$c_i = \Theta(1)$**

Amortized complexity of n insertions: direct (1/2)

- Start from an empty array T_0 (capacity 0) and insert **n elements**, giving arrays $T_1, \dots, T_n$ of sizes $1, \dots, n$; let c_i be the cost of the i -th insertion
- Insertion i is **expensive** when T_{i-1} is full: $i - 1 = \text{cap}(T_{i-1})$; the $i - 1$ elements are then copied, so **$c_i = \Theta(i)$** ; otherwise **$c_i = \Theta(1)$**
- Since the capacity doubles, the capacities taken are $0, 1, 2, 4, 8, 16, \dots$: insertion i is expensive iff $i - 1$ is 0 or a **power of 2**, i.e., $i \in \{1, 2, 3, 5, 9, 17, \dots\}$

Amortized complexity of n insertions: direct (1/2)

- Start from an empty array T_0 (capacity 0) and insert n elements, giving arrays $T_1, \dots, T_n$ of sizes $1, \dots, n$; let c_i be the cost of the i -th insertion
- Insertion i is **expensive** when T_{i-1} is full: $i - 1 = \text{cap}(T_{i-1})$; the $i - 1$ elements are then copied, so $c_i = \Theta(i)$; otherwise $c_i = \Theta(1)$
- Since the capacity doubles, the capacities taken are $0, 1, 2, 4, 8, 16, \dots$: insertion i is expensive iff $i - 1$ is 0 or a **power of 2**, i.e., $i \in \{1, 2, 3, 5, 9, 17, \dots\}$
- Among the n insertions, the expensive ones are $i = 1$ and $i = 2^k + 1$ for $0 \leq k \leq \lfloor \log_2(n - 1) \rfloor$: **$\Theta(\log n)$** of them, with $c_{2^k+1} = \Theta(2^k)$

Amortized complexity of n insertions: direct (1/2)

- Start from an empty array T_0 (capacity 0) and insert **n elements**, giving arrays $T_1, \dots, T_n$ of sizes $1, \dots, n$; let c_i be the cost of the i -th insertion
- Insertion i is **expensive** when T_{i-1} is full: $i - 1 = \text{cap}(T_{i-1})$; the $i - 1$ elements are then copied, so **$c_i = \Theta(i)$** ; otherwise **$c_i = \Theta(1)$**
- Since the capacity doubles, the capacities taken are $0, 1, 2, 4, 8, 16, \dots$: insertion i is expensive iff $i - 1$ is 0 or a **power of 2**, i.e., $i \in \{1, 2, 3, 5, 9, 17, \dots\}$
- Among the n insertions, the expensive ones are $i = 1$ and $i = 2^k + 1$ for $0 \leq k \leq \lfloor \log_2(n - 1) \rfloor$: **$\Theta(\log n)$** of them, with $c_{2^k+1} = \Theta(2^k)$
- All the others cost $\Theta(1)$: $O(n)$ in total

Amortized complexity of n insertions: direct (2/2)

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} \left(O(n) + \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} \Theta(2^k) \right)$$

Amortized complexity of n insertions: direct (2/2)

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} \left(O(n) + \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} \Theta(2^k) \right)$$

Recall: $\sum_{k=0}^m \alpha^k = \frac{\alpha^{m+1} - 1}{\alpha - 1} \quad (\alpha \neq 1), \quad \text{so} \quad \sum_{k=0}^m 2^k = 2^{m+1} - 1 < 2 \times 2^m$

Amortized complexity of n insertions: direct (2/2)

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} \left(O(n) + \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} \Theta(2^k) \right)$$

Recall: $\sum_{k=0}^m \alpha^k = \frac{\alpha^{m+1} - 1}{\alpha - 1} \quad (\alpha \neq 1), \quad \text{so} \quad \sum_{k=0}^m 2^k = 2^{m+1} - 1 < 2 \times 2^m$

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} \left(O(n) + O\left(2^{\lfloor \log_2(n-1) \rfloor}\right) \right) \leq \frac{1}{n} (O(n) + O(n)) = \mathbf{O(1)}$$

Amortized complexity of n insertions: direct (2/2)

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} \left(O(n) + \sum_{k=0}^{\lfloor \log_2(n-1) \rfloor} \Theta(2^k) \right)$$

Recall: $\sum_{k=0}^m \alpha^k = \frac{\alpha^{m+1} - 1}{\alpha - 1} \quad (\alpha \neq 1), \quad \text{so} \quad \sum_{k=0}^m 2^k = 2^{m+1} - 1 < 2 \times 2^m$

$$\frac{1}{n} \sum_{i=1}^n c_i \leq \frac{1}{n} \left(O(n) + O\left(2^{\lfloor \log_2(n-1) \rfloor}\right) \right) \leq \frac{1}{n} (O(n) + O(n)) = \mathbf{O(1)}$$

The total cost of n insertions is $O(n)$: the copies, however expensive individually, sum to less than $2n$ element moves

Amortized complexity of n insertions: potential (1/2)

- ★ Same sequence: from an empty T_0 , insert n elements, giving $T_1, \dots, T_n$

Amortized complexity of n insertions: potential (1/2)

- ★ Same sequence: from an empty T_0 , insert n elements, giving $T_1, \dots, T_n$
- ★ Define the potential of an array T of size $n(T)$ and capacity $cap(T)$ as

$$\Phi(T) := \alpha \times (2n(T) - cap(T))$$

for a constant $\alpha > 0$ to be chosen

Amortized complexity of n insertions: potential (1/2)

- ★ Same sequence: from an empty T_0 , insert n elements, giving $T_1, \dots, T_n$
- ★ Define the potential of an array T of size $n(T)$ and capacity $cap(T)$ as

$$\Phi(T) := \alpha \times (2n(T) - cap(T))$$

for a constant $\alpha > 0$ to be chosen

- ★ Intuition: right after a doubling, the array is half full and $\Phi = 0$; each cheap insertion then adds 2α to the reserve; when the array is full, $n(T) = cap(T)$ and the reserve $\alpha n(T)$ pays for copying the $n(T)$ elements

Amortized complexity of n insertions: potential (1/2)

- ★ Same sequence: from an empty T_0 , insert n elements, giving $T_1, \dots, T_n$
- ★ Define the potential of an array T of size $n(T)$ and capacity $cap(T)$ as

$$\Phi(T) := \alpha \times (2n(T) - cap(T))$$

for a constant $\alpha > 0$ to be chosen

- ★ Intuition: right after a doubling, the array is half full and $\Phi = 0$; each cheap insertion then adds 2α to the reserve; when the array is full, $n(T) = cap(T)$ and the reserve $\alpha n(T)$ pays for copying the $n(T)$ elements
- ★ $\Phi(T_0) = 0$ and $\Phi(T_i) \geq 0$ for all i , since the array is always at least half full: the potential method applies

Amortized complexity of n insertions: potential (1/2)

- ★ Same sequence: from an empty T_0 , insert n elements, giving $T_1, \dots, T_n$
- ★ Define the potential of an array T of size $n(T)$ and capacity $cap(T)$ as

$$\Phi(T) := \alpha \times (2n(T) - cap(T))$$

for a constant $\alpha > 0$ to be chosen

- ★ Intuition: right after a doubling, the array is half full and $\Phi = 0$; each cheap insertion then adds 2α to the reserve; when the array is full, $n(T) = cap(T)$ and the reserve $\alpha n(T)$ pays for copying the $n(T)$ elements
- ★ $\Phi(T_0) = 0$ and $\Phi(T_i) \geq 0$ for all i , since the array is always at least half full: the potential method applies
- ★ It remains to bound the amortized cost $\hat{c}_i = c_i + \Phi(T_i) - \Phi(T_{i-1})$ of each insertion

Amortized complexity of n insertions: potential (2/2)

- **Cheap insertion** ($i - 1 < \text{cap}(T_{i-1})$): the capacity is unchanged and the size increases by 1, so $\Phi(T_i) - \Phi(T_{i-1}) = 2\alpha$, and $c_i \leq \alpha$ if α is chosen larger than the constant cost of writing one element

Amortized complexity of n insertions: potential (2/2)

- **Cheap insertion** ($i - 1 < \text{cap}(T_{i-1})$): the capacity is unchanged and the size increases by 1, so $\Phi(T_i) - \Phi(T_{i-1}) = 2\alpha$, and $c_i \leq \alpha$ if α is chosen larger than the constant cost of writing one element
- **Expensive insertion** ($i - 1 = \text{cap}(T_{i-1}), i \geq 2$): the capacity goes from $i - 1$ to $2(i - 1)$, so

$$\Phi(T_{i-1}) = \alpha(2(i - 1) - (i - 1)) = \alpha(i - 1), \quad \Phi(T_i) = \alpha(2i - 2(i - 1)) = 2\alpha,$$

hence $\Phi(T_i) - \Phi(T_{i-1}) = \alpha(3 - i)$; and $c_i \leq \alpha i$ if α is also chosen larger than the cost of copying one element

Amortized complexity of n insertions: potential (2/2)

- + **Cheap insertion** ($i - 1 < \text{cap}(T_{i-1})$): the capacity is unchanged and the size increases by 1, so $\Phi(T_i) - \Phi(T_{i-1}) = 2\alpha$, and $c_i \leq \alpha$ if α is chosen larger than the constant cost of writing one element
- + **Expensive insertion** ($i - 1 = \text{cap}(T_{i-1})$, $i \geq 2$): the capacity goes from $i - 1$ to $2(i - 1)$, so

$$\Phi(T_{i-1}) = \alpha(2(i - 1) - (i - 1)) = \alpha(i - 1), \quad \Phi(T_i) = \alpha(2i - 2(i - 1)) = 2\alpha,$$

hence $\Phi(T_i) - \Phi(T_{i-1}) = \alpha(3 - i)$; and $c_i \leq \alpha i$ if α is also chosen larger than the cost of copying one element

- + In both cases:

$$\hat{c}_i \leq \max(\alpha + 2\alpha, \alpha i + \alpha(3 - i)) = 3\alpha = O(1)$$

(and $\hat{c}_1 \leq 2\alpha$ for the very first insertion), so the amortized complexity of an insertion is $O(1)$

Removal in dynamic arrays

- ★ As presented: removal of the last element in $O(1)$, by decrementing n

Removal in dynamic arrays

- ★ As presented: removal of the last element in $O(1)$, by decrementing n
- ★ But then **memory is never released**, even if the array shrinks a lot: poor space complexity

Removal in dynamic arrays

- ✦ As presented: removal of the last element in $O(1)$, by decrementing n
- ✦ But then **memory is never released**, even if the array shrinks a lot: poor space complexity
- ✦ Also, the potential $\alpha(2n - \text{cap})$ becomes negative when the array is less than half full: the analysis above does not cover sequences mixing insertions and removals

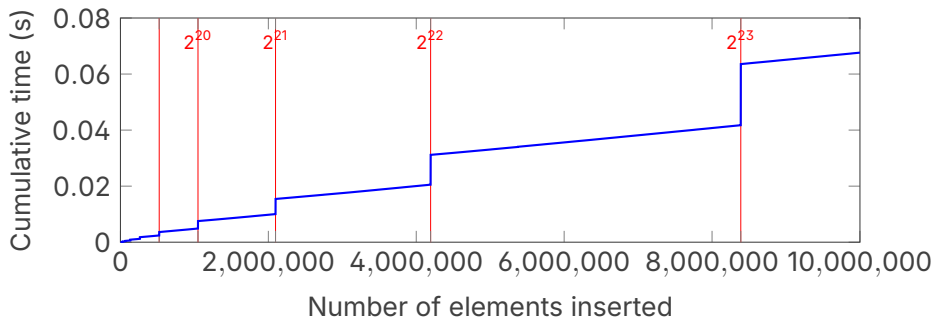
Removal in dynamic arrays

- ★ As presented: removal of the last element in $O(1)$, by decrementing n
- ★ But then **memory is never released**, even if the array shrinks a lot: poor space complexity
- ★ Also, the potential $\alpha(2n - \text{cap})$ becomes negative when the array is less than half full: the analysis above does not cover sequences mixing insertions and removals
- ★ Natural idea: halve the capacity when the array becomes half empty; **wrong**: alternating insertions and removals around the threshold would copy the array at each operation

Removal in dynamic arrays

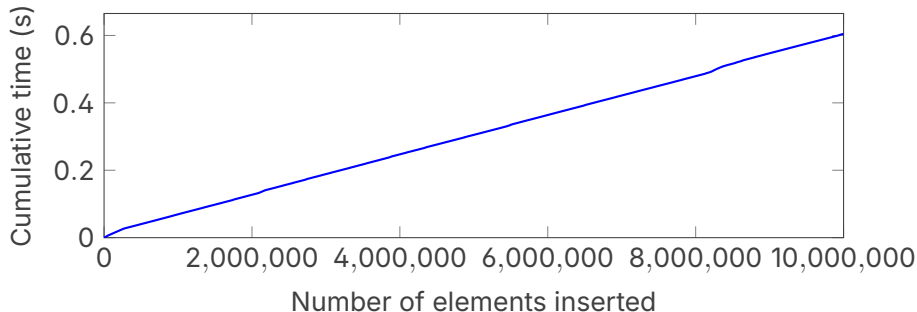
- As presented: removal of the last element in $O(1)$, by decrementing n
- But then **memory is never released**, even if the array shrinks a lot: poor space complexity
- Also, the potential $\alpha(2n - \text{cap})$ becomes negative when the array is less than half full: the analysis above does not cover sequences mixing insertions and removals
- Natural idea: halve the capacity when the array becomes half empty; **wrong**: alternating insertions and removals around the threshold would copy the array at each operation
- In practice**: halve the capacity when the array is only a **quarter** full; with the potential $\alpha |2n - \text{cap}|$, any sequence of insertions and removals costs $O(1)$ **amortized** per operation, and the array is always at least a quarter full: full analysis in Cormen et al. [2022, Section 16.4]

Measured: 10^7 insertions in a `std::vector`



g++ -O2, on a laptop; the cumulative time is **linear** in the number of insertions, i.e., amortized $O(1)$; the **steps** are the reallocations, at the powers of 2 (red lines)

Measured: 10^7 insertions in a Python list



CPython 3.12, same machine; also linear, with a constant factor of about 10 compared to C++, and steps too small to be seen: the growth factor is smaller, so reallocations are more frequent but each copies less

Coda



Wednesday's tutorial and references

- ★ Wednesday's tutorial: arrays and pointers in C (a function modifying an array, two-dimensional arrays), dynamic allocation with `malloc/free` checked with AddressSanitizer, the same programs with `std::vector`, `std::string`, and `std::map` in C++, and implementing a dynamic array with doubling in C, timing n insertions to observe the amortized $O(1)$
- ★ Next lecture: object-oriented programming, and `struct` in C, to bundle the pointer, size, and capacity of a dynamic array into one value
- ★ References: King [2008], chapters 8 (arrays), 11 and 12 (pointers, pointers and arrays), and 17 (dynamic storage allocation); Cormen et al. [2022], chapter 16 (amortized analysis, with dynamic tables in Section 16.4); the C++ containers library in cppreference.com [2026]

Licensing

This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.



Bibliography I

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
Introduction to Algorithms. MIT Press, 4th edition, 2022. ISBN 978-0-262-04630-5.
URL <https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/>.
- cppreference.com. Containers library. <https://en.cppreference.com/w/cpp/container>, 2026.
- K. N. King. *C Programming: A Modern Approach*. W.W. Norton & Company, 2 edition, 2008. ISBN 0393979504. URL <http://knking.com/books/c2/index.html>.