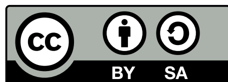
Memory Management, C/C++ Collections, & Amortized Complexity

The Art of Computer Programming

Pierre Senellart



13 October 2025

Outline

C Collections

Memory Management

Amortized Complexity

Collections in C++

Dynamic Array

Collections in C

- The C standard library provides very limited support for collections
- Essentially, only fixed-size arrays
- No built-in support for dynamic arrays, sets, or associative arrays
- Any more advanced collection requires manual implementation or using third-party libraries

C Arrays

- Declaration example: `int a[10];` creates an array of 10 integers
- Contiguous area of memory: elements are stored one after each other in memory
- Fixed size: impossible to add or remove elements
- Mutable unless specified constant (see further)
- Arrays decay to pointers when passed to functions, i.e., the array variable is replaced to a pointer to its first element
  - Example: `void f(int a[]) { ... }` is equivalent to `void f(int *a) { ... }`
  - You lose information about array length
- No bounds checking: attempts to access a value beyond the size of the array will result in undefined behavior (e.g., crash, access to another part of the memory, or other unpredictable behavior)

Constants in C

- `const` allows you to declare data as constant, i.e., read-only

C

```c
const int a = 42; // a cannot be modified
const int b[10]={0,1,2,3,4,5,6,7,8,9}; // immutable array of
↪ integers
```

- Can also apply to pointers (read right-to-left):
  - `const int *p;` pointer to constant integer
  - `int * const p;` constant pointer to integer
  - `const int * const p;` constant pointer to constant integer
- Using `const` helps prevent accidental modification and clarifies intent

Constants in C++

- `const` works as in C: it declares data as read-only

C++
```cpp
const int a = 42; // a cannot be modified
const int b[10]={0,1,2,3,4,5,6,7,8,9}; // immutable array of
↪  integers
```

- Can also apply to references:
  - `void f(int x);` parameter passed by value
  - `void f(int& x);` parameter passed by reference, can be modified
  - `void f(const int& x);` parameter passed by reference, cannot be modified
- Use `const` reference for function parameters that you do not want to modify but are large and would be too costly to copy (anything that is not a primitive type)

The sizeof Operator in C/C++

- `sizeof` returns the size (in bytes) of a type, a variable, or an expression of that type
- Examples:

C

```c
int a[10];
printf("%zu\n", sizeof(int));          // size of int type
printf("%zu\n", sizeof(a[0]));         // size of one element
printf("%zu\n", sizeof(a));            // size of the array in bytes
```

- When applied to an array variable, it gives the total size of the array
- Warning: if the array has decayed to a pointer (e.g., passed to a function), `sizeof(a)` returns the size of the pointer, not the array
- Can also be used with types directly: `sizeof(double)`, `sizeof(char[20])`
- Useful for computing array length:

$$\text{length} = \frac{\texttt{sizeof(array)}}{\texttt{sizeof(array[0])}}$$

Pointer Arithmetic in C/C++

- In C/C++, arrays decay to pointers when passed to functions (or when used as a pointer)
- Accessing an element via `a[i]` is equivalent to writing `*(a + i)`: $\Theta(1)$!

C

```c
int a[5] = {10, 20, 30, 40, 50};
int *p = a;
printf("%d\n", a[3]);   // prints 40
printf("%d\n", *(p+3)); // also prints 40
```

- You can use pointer arithmetic directly:
  - Increment pointer: `p++` points to next element
  - Access element via offset: `*(p+2)`
- Be careful: pointer arithmetic respects type size: `p+1` advances by `sizeof(T)` bytes if `p` is a pointer of type `T*`

NULL and nullptr in C and C++

- NULL (C) and nullptr (C++) represent null pointers, i.e., pointers that point to no valid memory location
- Can be assigned to any pointer type to indicate “no value”
- In C, `NULL` is typically defined as 0 of type `void*`
- In C++, prefer `nullptr` (of type `std::nullptr_t`)
- Both are automatically interpreted as `true`/`false` Booleans for use in tests

C

```c
int *p = NULL;
/* ... */
if (p) { *p = 42; /* safe only if not NULL */ }
```

C++

```cpp
int *q = nullptr;
/* ... */
if (q) { *q = 42; /* safe only if not nullptr */ }
```

C Arrays: Common Operations

Type	`T a[N];` Array of type T and size N	
Initialization	`int a[3] = {0, 1, 2};` or `int a[] = {0, 1, 2};` to automatically compute size	
Access element	`a[i]`	O(1)
Modify element	`a[i] = value;`	O(1)
Length	`sizeof(a)/sizeof(T)` but not possible if array was passed to function	O(1)
Loop		O(N)

C

```c
for(int i=0; i<N; i++) {
  // use a[i]
}
```

Example Use of Array in Function

C

```c
int sum_array(const int *a, int n)
{
  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, sizeof(a)/sizeof(int)));
  return 0;
}
```

Outline

C Collections

Memory Management

Amortized Complexity

Collections in C++

Dynamic Array

Beyond the Stack: Other Memory Regions

- So far, we have only used local variables (local to a block in C/C++, to a function in Python) as well as function parameters

Beyond the Stack: Other Memory Regions

- So far, we have only used local variables (local to a block in C/C++, to a function in Python) as well as function parameters
- The content of local variables and function parameters are stored in a specific memory region: the stack (also used to store return addresses of function calls)

Beyond the Stack: Other Memory Regions

- So far, we have only used local variables (local to a block in C/C++, to a function in Python) as well as function parameters
- The content of local variables and function parameters are stored in a specific memory region: the stack (also used to store return addresses of function calls)
- The stack has limited size

Beyond the Stack: Other Memory Regions

- So far, we have only used local variables (local to a block in C/C++, to a function in Python) as well as function parameters
- The content of local variables and function parameters are stored in a specific memory region: the stack (also used to store return addresses of function calls)
- The stack has limited size
- How to store values of arbitrary size (as long as available memory allows)?

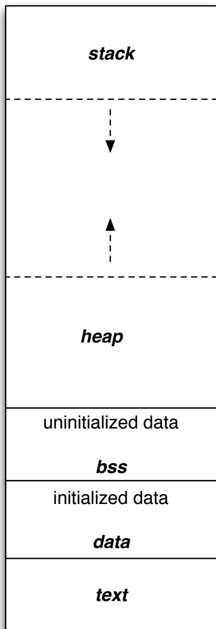
Beyond the Stack: Other Memory Regions

- So far, we have only used local variables (local to a block in C/C++, to a function in Python) as well as function parameters
- The content of local variables and function parameters are stored in a specific memory region: the stack (also used to store return addresses of function calls)
- The stack has limited size
- How to store values of arbitrary size (as long as available memory allows)?
- How to have data created within a function whose extent (lifetime) runs beyond the end of a function?

Beyond the Stack: Other Memory Regions

- So far, we have only used local variables (local to a block in C/C++, to a function in Python) as well as function parameters
- The content of local variables and function parameters are stored in a specific memory region: the stack (also used to store return addresses of function calls)
- The stack has limited size
- How to store values of arbitrary size (as long as available memory allows)?
- How to have data created within a function whose extent (lifetime) runs beyond the end of a function?
- How to have variables accessible across functions?

Memory of a Process



The memory of a running program (a process) is formed of different regions (also called segments):

- **stack** for local variables, function parameters, return addresses of function calls
- **heap** for dynamically allocated data
- **bss** for uninitialized global variables, i.e., global variables with no specified value in the code
- **data** for initialized global variables, as well as other values stored within the code
- **text** for the program code itself (as binary machine code)

Global Variables

- Variables accessible across functions and function calls, with one fixed memory address throughout the entire duration of a program and a value that can be read and written to across functions
- Can be:
  - assigned an explicit value at declaration: initialized global variable, end up in data
  - or not: uninitialized global variables, end up in bss – they typically get a default value when the program is started (e.g., 0 for integer types in C/C++)
- Common for constants (often written in all-uppercase letters)
- For non-constants, use only when absolutely required, these non-local variables make the program harder to read and maintain, and pose problems in parallel computation settings
- Global variables are stored in data or bss, not on the stack or heap

Global Variables in C/C++

- Simply define the variable using a standard uninitialized declaration (*type name;*) or initialized declaration (*type name = value;*) outside of a function
- These variables can be freely used in any functions that follow this definition (unless hidden by a local variable with the same name)

C

```c
// Global constant (stored in data segment)
const double PI = 3.14159265358979323846264338327 9;

double perimeter(double radius)
{
  return 2 * PI * radius;
}
```

Static Variables in C/C++

In C/C++, a static variable (declared with `static`) is the same as a global variable, except its scope is limited to the block it is declared in (use sparingly!)

C

```c
#include <stdio.h>

void counter()
{
  static int count = 0; // initialized once, kept in data segment
  count++;
  printf("Count = %d\n", count);
}

int main(void)
{
  counter(); // prints 1
  counter(); // prints 2
  return 0;
}
```

Global Variables in Python

- Variables outside of a function are global to that file
- To modify a global variable inside a function, use the `global` keyword (to use it read-only, simply refer to it by name)

Python

```
PI = 3.141592653589793
def perimeter(radius):
    return 2 * PI * radius

count = 0
def increment():
    global count
    count += 1

print(perimeter(3)) # prints 18.84955592153876
increment()
increment()
print(count) # prints 2
```

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure
- Sometimes we need to create variables whose size or lifetime is decided at runtime

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure
- Sometimes we need to create variables whose size or lifetime is decided at runtime
- Such variables are stored in a different region of memory: the heap

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure
- Sometimes we need to create variables whose size or lifetime is decided at runtime
- Such variables are stored in a different region of memory: the heap
- They are created and destroyed explicitly by the program.

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure
- Sometimes we need to create variables whose size or lifetime is decided at runtime
- Such variables are stored in a different region of memory: the heap
- They are created and destroyed explicitly by the program.
- Common use cases:

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure
- Sometimes we need to create variables whose size or lifetime is decided at runtime
- Such variables are stored in a different region of memory: the heap
- They are created and destroyed explicitly by the program.
- Common use cases:
  - Variable-size data (e.g., variable-size arrays, sets, associative arrays)

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure
- Sometimes we need to create variables whose size or lifetime is decided at runtime
- Such variables are stored in a different region of memory: the heap
- They are created and destroyed explicitly by the program.
- Common use cases:
  - Variable-size data (e.g., variable-size arrays, sets, associative arrays)
  - Values that outlive the function that created them

Dynamically Allocated Variables

- So far: local (stack), global and static (data/bss) variables — all with fixed size and lifetime determined by the program's structure
- Sometimes we need to create variables whose size or lifetime is decided at runtime
- Such variables are stored in a different region of memory: the heap
- They are created and destroyed explicitly by the program.
- Common use cases:
  - Variable-size data (e.g., variable-size arrays, sets, associative arrays)
  - Values that outlive the function that created them
- Added complexity: the programmer (or the system) must ensure correct allocation (when the value is created) and release (when the value is not needed any more) of heap memory

Memory Management

- Memory management: controlling allocation and release (deallocation) of heap memory
- Two main approaches:

  Explicit management the programmer decides when to allocate and free memory (C, C++)

  Automatic management the runtime system reclaims memory no longer referenced by the program (garbage collection, often used in interpreted languages, such as Python)
- Explicit management offers:
  - Fine control and performance
  - But risks of memory leaks, dangling pointers, and double release
- Garbage collection offers:
  - Ease of use and safety
  - But much less predictable performance and memory usage

Memory Allocation in C and C++

- In C: memory is allocated with `malloc` (for a value of a fixed size) or `calloc` (for n values of a fixed size) – returns NULL (equal to 0) if not possible
- In C++: memory is allocated using the `new` operator (can take an initialization value); arrays of values are allocated using the `new[]` operator – throws an exception if not possible
- Allocation requests a block of bytes on the heap and returns a pointer to it
- The program must later release this memory manually.

C

```c
#include <stdlib.h>

int *pi = malloc(sizeof(int));      // memory for an int  on the heap
int *ai = calloc(10, sizeof(int)); // memory for 10 ints on the heap
if(!pi || !ai) { /* handle allocation failure */ }
```

C++

```cpp
double *pd = new double(3.14); // memory for  a double, set to 3.14
double *ad = new double[24];   // memory for 24 doubles
```

Memory Release in C and C++

- Dynamically allocated memory must be explicitly released when no longer needed
- In C: use `free(ptr);`
- In C++: use `delete ptr;` or `delete[] ptr;` (depending on whether it was created with `new` or `new[]`)
- Failing to free memory leads to a memory leak
- Freeing memory twice or using a pointer after freeing it causes undefined behavior

C

```c
free(pi);
free(ai);
```

C++

```cpp
delete pd;
delete [] ad;
```

Undefined Behavior in C and C++

- Undefined behavior occurs when the C/C++ standard does not prescribe what should happen for a certain operation
- Common causes related to memory management:
  - Using a pointer after it has been freed (dangling pointer)
  - Freeing the same memory block twice (double free)
  - Accessing memory outside the bounds of an array (out-of-bound error)
  - Dereferencing uninitialized or null pointers
- Undefined behavior can result in:
  - Program crashes
  - Silent data corruption
  - Security vulnerabilities
  - Code that behaves differently on different compilers or runs
- Best practices: always initialize pointers, carefully manage allocation/deallocation and pointer manipulation

What About Python?

- Python uses automatic memory management:
  - Most values are actually automatically allocated on the heap (local variables just put the address of this heap value on the stack)
  - The programmer never calls `malloc` or `free`
- Memory is reclaimed automatically by Python's garbage collector, based mainly on reference counting
- When an object's reference count drops to zero, its memory can be released (when the garbage collector gets around to it)
- This makes programming easier and safer — but at the cost of less predictable timing of deallocation

Python

```python
a = [1, 2, 3] # allocates heap memory to store this list, address in a
b = a # both refer to the same list (same address), with 2 references
# memory freed automatically when neither a nor b exists anymore
```

Outline

C Collections

Memory Management

Amortized Complexity

Collections in C++

Dynamic Array

Amortized Complexity

- So far, algorithmic (time) complexity has been measured separately for each operation of a data structure

Amortized Complexity

- So far, algorithmic (time) complexity has been measured separately for each operation of a data structure
- Sometimes, it is impossible to give a good bound on the time of each individual operation, but it is possible to bound the average time of a finite sequence of operations

Amortized Complexity

- So far, algorithmic (time) complexity has been measured separately for each operation of a data structure
- Sometimes, it is impossible to give a good bound on the time of each individual operation, but it is possible to bound the average time of a finite sequence of operations
- Let $o_1, \ldots, o_n$ be a sequence of n operations on a data structure, with costs $c_1, \ldots, c_n$

Amortized Complexity

- So far, algorithmic (time) complexity has been measured separately for each operation of a data structure
- Sometimes, it is impossible to give a good bound on the time of each individual operation, but it is possible to bound the average time of a finite sequence of operations
- Let $o_1, \ldots, o_n$ be a sequence of n operations on a data structure, with costs $c_1, \ldots, c_n$
- The average complexity of an operation o_i is

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

Amortized Complexity

- So far, algorithmic (time) complexity has been measured separately for each operation of a data structure
- Sometimes, it is impossible to give a good bound on the time of each individual operation, but it is possible to bound the average time of a finite sequence of operations
- Let $o_1, \ldots, o_n$ be a sequence of n operations on a data structure, with costs $c_1, \ldots, c_n$
- The average complexity of an operation o_i is

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

- This is called the amortized complexity

Amortized Complexity

- So far, algorithmic (time) complexity has been measured separately for each operation of a data structure
- Sometimes, it is impossible to give a good bound on the time of each individual operation, but it is possible to bound the average time of a finite sequence of operations
- Let $o_1, \ldots, o_n$ be a sequence of n operations on a data structure, with costs $c_1, \ldots, c_n$
- The average complexity of an operation o_i is

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

- This is called the amortized complexity
- It only makes sense if we define precisely the type of sequence of operations considered (e.g., insertions)

Amortized Complexity vs Average Complexity

- Two orthogonal notions:

Amortized Complexity vs Average Complexity

- Two orthogonal notions:

  Amortized complexity: On average, how long an operation takes within a finite sequence of operations of a given type

Amortized Complexity vs Average Complexity

- Two orthogonal notions:

  Amortized complexity On average, how long an operation takes within a finite sequence of operations of a given type

  Average-case complexity On average, how long an operation takes, over all possible inputs (as opposed to worst-case)

Amortized Complexity vs Average Complexity

- Two orthogonal notions:

  Amortized complexity
  : On average, how long an operation takes within a finite sequence of operations of a given type

  Average-case complexity
  : On average, how long an operation takes, over all possible inputs (as opposed to worst-case)

- Therefore, we can talk about amortized worst-case complexity (this is the usual notion) and amortized average-case complexity

Computing Amortized Complexity

- Manually, by computing the cost of each operation in a sequence and averaging them

Computing Amortized Complexity

- Manually, by computing the cost of each operation in a sequence and averaging them
- Sometimes difficult because it requires precise calculation and considering the impact of each operation on subsequent ones

Computing Amortized Complexity

- Manually, by computing the cost of each operation in a sequence and averaging them
- Sometimes difficult because it requires precise calculation and considering the impact of each operation on subsequent ones
- Trick: associate a potential to the data structure. When an operation is cheap, increase the potential; when the operation is expensive, compensate by reducing the potential.

Computing Amortized Complexity

- Manually, by computing the cost of each operation in a sequence and averaging them
- Sometimes difficult because it requires precise calculation and considering the impact of each operation on subsequent ones
- Trick: associate a potential to the data structure. When an operation is cheap, increase the potential; when the operation is expensive, compensate by reducing the potential.
- The potential method allows one to consider each operation individually: the effect on subsequent operations is replaced by the effect on the potential

Computing Amortized Complexity

- Manually, by computing the cost of each operation in a sequence and averaging them
- Sometimes difficult because it requires precise calculation and considering the impact of each operation on subsequent ones
- Trick: associate a potential to the data structure. When an operation is cheap, increase the potential; when the operation is expensive, compensate by reducing the potential.
- The potential method allows one to consider each operation individually: the effect on subsequent operations is replaced by the effect on the potential
- But sometimes it is hard to find the right potential function

Potential Method

- Associate a potential $\Phi(X)$ (real number) to a data structure X

Potential Method

- Associate a potential $\Phi(X)$ (real number) to a data structure X
- Let $o_1, \ldots, o_n$ be a sequence of n operations with real costs $c_1, \ldots, c_n$, and corresponding structures $X_0, \ldots, X_n$ ($X_i = o_i(X_{i-1})$)

Potential Method

- Associate a potential $\Phi(X)$ (real number) to a data structure X
- Let $o_1, \ldots, o_n$ be a sequence of n operations with real costs $c_1, \ldots, c_n$, and corresponding structures $X_0, \ldots, X_n$ ($X_i = o_i(X_{i-1})$)
- Define $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$

Potential Method

- Associate a potential $\Phi(X)$ (real number) to a data structure X
- Let $o_1, \ldots, o_n$ be a sequence of n operations with real costs $c_1, \ldots, c_n$, and corresponding structures $X_0, \ldots, X_n$ ($X_i = o_i(X_{i-1})$)
- Define $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$
- Then: $\frac{1}{n} \sum_{i=1}^{n} \hat{c}_i = \frac{1}{n} \sum_{i=1}^{n} c_i + \frac{1}{n}(\Phi(X_n) - \Phi(X_0))$

Potential Method

- Associate a potential $\Phi(X)$ (real number) to a data structure X
- Let $o_1, \ldots, o_n$ be a sequence of n operations with real costs $c_1, \ldots, c_n$, and corresponding structures $X_0, \ldots, X_n$ $(X_i = o_i(X_{i-1}))$
- Define $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$
- Then: $\frac{1}{n} \sum_{i=1}^{n} \hat{c}_i = \frac{1}{n} \sum_{i=1}^{n} c_i + \frac{1}{n}(\Phi(X_n) - \Phi(X_0))$
- If $\hat{c}_i = O(f(|X_n|))$, then $\frac{1}{n} \sum_{i=1}^{n} \hat{c}_i = O(f(|X_n|))$

Potential Method

- Associate a potential $\Phi(X)$ (real number) to a data structure X
- Let $o_1, \ldots, o_n$ be a sequence of n operations with real costs $c_1, \ldots, c_n$, and corresponding structures $X_0, \ldots, X_n$ ($X_i = o_i(X_{i-1})$)
- Define $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$
- Then: $\frac{1}{n}\sum_{i=1}^{n} \hat{c}_i = \frac{1}{n}\sum_{i=1}^{n} c_i + \frac{1}{n}(\Phi(X_n) - \Phi(X_0))$
- If $\hat{c}_i = O(f(|X_n|))$, then $\frac{1}{n}\sum_{i=1}^{n} \hat{c}_i = O(f(|X_n|))$
- So $\frac{1}{n}\sum_{i=1}^{n} c_i = O(f(|X_n|))$ as long as, for all n, $\Phi(X_n) - \Phi(X_0) \geqslant 0$

Potential Method

- Associate a potential $\Phi(X)$ (real number) to a data structure X
- Let $o_1, \ldots, o_n$ be a sequence of n operations with real costs $c_1, \ldots, c_n$, and corresponding structures $X_0, \ldots, X_n$ $(X_i = o_i(X_{i-1}))$
- Define $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$
- Then: $\frac{1}{n}\sum_{i=1}^{n} \hat{c}_i = \frac{1}{n}\sum_{i=1}^{n} c_i + \frac{1}{n}(\Phi(X_n) - \Phi(X_0))$
- If $\hat{c}_i = O(f(|X_n|))$, then $\frac{1}{n}\sum_{i=1}^{n} \hat{c}_i = O(f(|X_n|))$
- So $\frac{1}{n}\sum_{i=1}^{n} c_i = O(f(|X_n|))$ as long as, for all n, $\Phi(X_n) - \Phi(X_0) \geqslant 0$
- Often take X_0 as the empty data structure and $\Phi(X_0) := 0$, giving the condition $\Phi(X_n) \geqslant 0$

Outline

C Collections

Memory Management

Amortized Complexity

Collections in C++

Dynamic Array

C++ Namespaces

- Namespaces are used in C++ to organize the standard library, third-party code, and own code
- In particular, many collections and utilities are defined in the std namespace: `std::vector`, `std::map`, `std::cout`, etc.
- You can refer to them with the fully qualified name:

C++

```cpp
std::vector<int> v;
std::cout << "Hello, world!" << std::endl;
```

- To avoid repeatedly writing `std::`, you can also bring names into scope using `using namespace std;`

Bringing Names into Scope

C++

```
// Bring one name into scope
using std::vector;
vector<int> v;

// Bring many names into scope
using std::cout;
using std::endl;

cout << "Hello" << endl;

// Bring all names from std into scope
using namespace std;
vector<int> v2;
cout << "Hi!" << endl;
```

Collections in C++

- C++ collections in C++ are defined in the `std` namespace
- To use collection `foobar`, you need `#include <foobar>`
- Collections are templates (generic types): the type of elements must be specified
- Examples:
  - `std::vector<int>` — dynamic array of integers
  - `std::vector<std::string>` — dynamic array of strings
  - `std::set<double>` — set of doubles
  - `std::map<std::string, int>` — associative array from strings to integers
- STL collections are designed to be:
  - Efficient: fine control over memory and performance
  - Type-safe: all elements share the same type
  - Generic: reusable for any element type

Iterating over Collections

C++

```cpp
#include <vector>
#include <iostream>

std::vector<int> v = {1,2,3};
for(int x : v) {
  // Repeat the following for each element
  std::cout << x << "\n";
}
```

You can also use:

- `continue` to skip to the next iteration
- `break` to exit the loop early

Sequential Containers in C++

Name (within std)	Abstract Type	Mutable?	Size
`array<T,N>`	Random-access ordered sequence	Yes	Fixed
`string`	Random-access ordered sequence (characters)	Yes	Dynamic
`vector<T>`	Random-access ordered sequence	Yes	Dynamic
`list<T>`	No-RA ordered sequence	Yes	Dynamic
`set<T>`	Ordered set	Yes	Dynamic
`unordered_set<T>`	Unordered set	Yes	Dynamic
`map<K,V>`	Ordered associative array	Yes	Dynamic
`unordered_map<K,V>`	Unordered associative array	Yes	Dynamic

Accessing and Modifying Collections

C++

```cpp
#include <vector>
#include <map>
#include <set>

std::vector<int> v = {1,2,3};
v.push_back(4);              // Append at end
v[0] = 10;                   // Modify
v.pop_back();                // Remove last element

std::set<int> s;
s.insert(5);                 // Add element
s.erase(5);                  // Remove element

std::map<std::string, int> d;
d["a"] = 1;                  // Insert/update
int val = d["a"];            // Access
d.erase("a");                // Remove
```

Iteration Examples

C++

```cpp
// Vector
for(int x : v) { std::cout << x << "\n"; }

// Set
for(int x : s) { std::cout << x << "\n"; }

// Map
for(auto [key, value] : d) {
    std::cout << key << " -> " << value << "\n";
}
```

C++ Vectors (analogous to Python list)

Type	`std::vector<T>`	
Definition	Ordered sequence of variable size n, mutable elements	

Init.	`std::vector<int> v;            // empty`
	`std::vector<int> v2 = {1,2,3};  // initializer list`	

Access	`v[i]`	$O(1)$
Modification	`v[i] = 42`	$O(1)$
Append at end	`v.push_back(42)`	amortized $O(1)$
Insert elsewhere	`v.insert(v.begin()+i, x)`	$O(n)$
Remove at end	`v.pop_back()`	$O(1)$
Remove elsewhere	`v.erase(v.begin()+i)`	$O(n)$
Size	`v.size()`	$O(1)$
Loop	`for(auto& x : v)`	$O(n)$

C++ Strings (roughly like Python str)

Type	`std::string`	
Init.	`std::string s;`	`// empty`
	`std::string s = "Hello";`	`// initializer`
Definition	Ordered sequence of bytes, dynamic size, mutable	
Access	`s[i]`	$O(1)$
Modification	`s[i]='a'`	$O(1)$
Append	`s += "abc"`	amortized $O(1)$ per character
Length	`s.size()`	$O(1)$
Loop	`for(char c : s)`	$O(n)$

C++ Sets (like Python set)

Type	`std::set<T>` (ordered) or `std::unordered_set<T>` (unordered)	
Definition	Unordered (or ordered) collection of unique elements	
Init.	`std::set<int> s = {1,2,3};` `std::unordered_set<int> us;`	
Membership test	`s.find(x) != s.end()`	$O(\log n)$ for set, $O(1)$ amortized for unordered_set
Add	`s.insert(x)`	same complexity
Remove	`s.erase(x)`	same complexity
Size	`s.size()`	$O(1)$
Loop	`for(auto& x : s)`	$O(n)$

C++ Maps (like Python dict)

Type `std::map<Key,Value>` (ordered) or `std::unordered_map<Key,Value>` (unordered)

Definition Associative array mapping keys to values

Init.
```
std::map<std::string, int> m;
std::map<std::string, int> m2 = {{"a",1}, {"b",2}};
```

Access `m[key] = value`
$O(\log n)$ for `map`, $O(1)$ amortized for `unordered_map`

Insert / Update `m[key] = value` — same complexity

Remove `m.erase(key)` — same complexity

Size `m.size()` — $O(1)$

Loop `for(auto& [k,v] : m)` — $O(n)$

Outline

C Collections

Memory Management

Amortized Complexity

Collections in C++

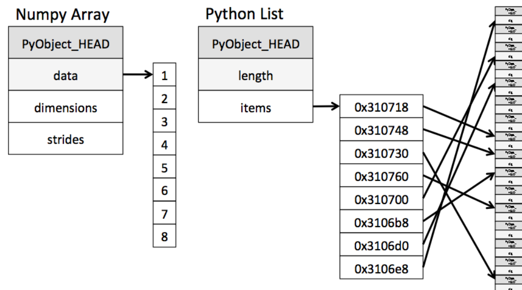
Dynamic Array

Dynamic Array Data Structure

- Data structure underlying Python lists and C++ vectors
- Pointer to a fixed-size array of size cap, along with integer $n \leqslant cap$ storing the number of elements actually used
- Random access in $\Theta(1)$ as with a fixed-size array
- Removing the last element in $\Theta(1)$ by decrementing n
- Insertion at the end in amortized complexity $\Theta(1)$ (see below)
- Removal or insertion elsewhere is $\Theta(n)$: array elements must be copied

Dynamic Array in Python

- Python lists are dynamic arrays
- Same for array.array from the standard library
- Python lists are heterogeneous: any Python value can be added to an existing list
- `array.array` (and fixed-size `numpy.array`) are homogeneous: more memory-efficient, faster data processing



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**
  create a new array T' of capacity $\max(2 \times cap, 1)$
  for $i \leftarrow 0$ **to** $n-1$ **do**
    $T'[i] \leftarrow T[i]$
  end for
  destroy the array referenced by T
  set T to point to T'
  $cap \leftarrow \max(2 \times cap, 1)$
end if
$T[n] \leftarrow x$
$n \leftarrow n+1$

Amortized Complexity of n Insertions – Direct (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$ of sizes $1 \ldots n$ and capacities $cap(T_1), \ldots, cap(T_n)$)

Amortized Complexity of n Insertions – Direct (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$ of sizes $1 \ldots n$ and capacities $cap(T_1), \ldots, cap(T_n)$)
- Compute c_i for $1 \leqslant i \leqslant n$

Amortized Complexity of n Insertions – Direct (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$ of sizes $1 \ldots n$ and capacities $cap(T_1), \ldots, cap(T_n)$)
- Compute c_i for $1 \leqslant i \leqslant n$
- For i such that $i - 1 \neq cap(T_{i-1})$, $c_i = \Theta(1)$

Amortized Complexity of n Insertions – Direct (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$ of sizes $1 \ldots n$ and capacities $cap(T_1), \ldots, cap(T_n)$)
- Compute c_i for $1 \leqslant i \leqslant n$
- For i such that $i - 1 \neq cap(T_{i-1})$, $c_i = \Theta(1)$
- For i such that $i - 1 = cap(T_{i-1})$, $c_i = \Theta(i)$

Amortized Complexity of n Insertions – Direct (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$ of sizes $1 \ldots n$ and capacities $cap(T_1), \ldots, cap(T_n)$)
- Compute c_i for $1 \leqslant i \leqslant n$
- For i such that $i - 1 \neq cap(T_{i-1})$, $c_i = \Theta(1)$
- For i such that $i - 1 = cap(T_{i-1})$, $c_i = \Theta(i)$
- Which i satisfy $i = cap(T_i)$? 0, 1, 2, 4, 8, 16, ... i.e., the powers of 2 (plus 0)

Amortized Complexity of n Insertions – Direct (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$ of sizes $1 \ldots n$ and capacities $cap(T_1), \ldots, cap(T_n)$)
- Compute c_i for $1 \leqslant i \leqslant n$
- For i such that $i - 1 \neq cap(T_{i-1})$, $c_i = \Theta(1)$
- For i such that $i - 1 = cap(T_{i-1})$, $c_i = \Theta(i)$
- Which i satisfy $i = cap(T_i)$? 0, 1, 2, 4, 8, 16, ... i.e., the powers of 2 (plus 0)
- There are $\lfloor \log_2(n-1) \rfloor + 1 = \Theta(\log n)$ powers of 2 between 0 and $n - 1$ (plus 0)

Amortized Complexity of n Insertions – Direct (2/2)

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

Amortized Complexity of n Insertions – Direct (2/2)

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

Recall: $\displaystyle\sum_{k=0}^{m} \alpha^k = \frac{\alpha^{m+1} - 1}{\alpha - 1} \quad (\alpha \neq 1)$

Amortized Complexity of n Insertions – Direct (2/2)

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

Recall: $$\sum_{k=0}^{m} \alpha^k = \frac{\alpha^{m+1}-1}{\alpha-1} \quad (\alpha \neq 1)$$

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

Amortized Complexity of n Insertions – Potential (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$)

Amortized Complexity of n Insertions – Potential (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$)
- Define the potential of array T_i of size i and capacity $cap(T_i)$ by $\Phi(T_i) := (2i - cap(T_i)) \times \alpha$ for some constant α

Amortized Complexity of n Insertions – Potential (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \ldots, T_n$)
- Define the potential of array T_i of size i and capacity $cap(T_i)$ by $\Phi(T_i) := (2i - cap(T_i)) \times \alpha$ for some constant α
- We have $\Phi(T_0) = 0$ and $\Phi(T_i) \geqslant 0$ for all i

Amortized Complexity of n Insertions – Potential (1/2)

- Start with empty array T_0, insert n elements (getting arrays $T_1, \dots, T_n$)
- Define the potential of array T_i of size i and capacity $cap(T_i)$ by
  $\Phi(T_i) := (2i - cap(T_i)) \times \alpha$ for some constant α
- We have $\Phi(T_0) = 0$ and $\Phi(T_i) \geqslant 0$ for all i
- Compute the amortized cost $\hat{c}_i$ via potential Φ: $\hat{c}_i := c_i + \Phi(X_i) - \Phi(X_{i-1})$

Amortized Complexity of n Insertions – Potential (2/2)

- For i such that $i - 1 \neq cap(T_{i-1})$, $c_i = O(1)$ and $\Phi(X_i) - \Phi(X_{i-1}) = 2\alpha$; choose α such that $c_i \leqslant \alpha$ for large i

Amortized Complexity of n Insertions – Potential (2/2)

- For i such that $i - 1 \neq cap(T_{i-1})$, $c_i = O(1)$ and $\Phi(X_i) - \Phi(X_{i-1}) = 2\alpha$; choose α such that $c_i \leqslant \alpha$ for large i
- For i such that $i - 1 = cap(T_{i-1})$, $c_i = O(i)$ and $\Phi(X_i) - \Phi(X_{i-1}) = (3 - i)\alpha$; choose α such that $c_i \leqslant \alpha i$ for large i

Amortized Complexity of n Insertions – Potential (2/2)

- For i such that $i - 1 \neq cap(T_{i-1})$, $c_i = O(1)$ and $\Phi(X_i) - \Phi(X_{i-1}) = 2\alpha$; choose α such that $c_i \leqslant \alpha$ for large i
- For i such that $i - 1 = cap(T_{i-1})$, $c_i = O(i)$ and $\Phi(X_i) - \Phi(X_{i-1}) = (3 - i)\alpha$; choose α such that $c_i \leqslant \alpha i$ for large i
- Then

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

Deletion in Dynamic Arrays

- As presented: deletion at the end in $O(1)$ (decrement n)

Deletion in Dynamic Arrays

- As presented: deletion at the end in $O(1)$ (decrement n)
- But this means we never free memory, even if the array shrinks significantly: excessive space complexity

Deletion in Dynamic Arrays

- As presented: deletion at the end in $O(1)$ (decrement n)
- But this means we never free memory, even if the array shrinks significantly: excessive space complexity
- Also: the analysis of a combination of insertions and deletions by the potential method is invalid if $2n - cap < 0$

Deletion in Dynamic Arrays

- As presented: deletion at the end in $O(1)$ (decrement n)
- But this means we never free memory, even if the array shrinks significantly: excessive space complexity
- Also: the analysis of a combination of insertions and deletions by the potential method is invalid if $2n - cap < 0$
- In practice: reduce array capacity when size decreases sufficiently

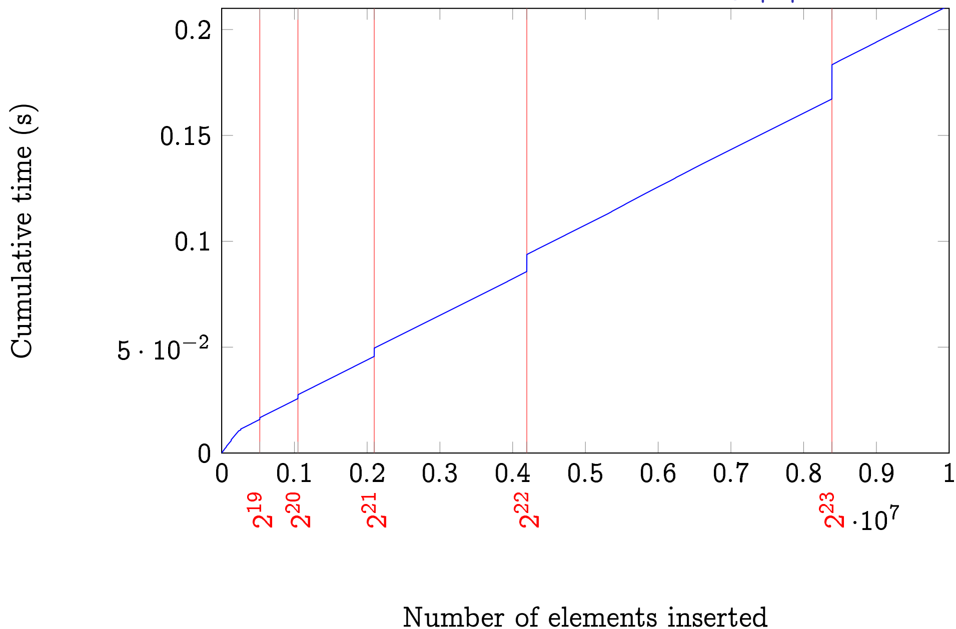
Deletion in Dynamic Arrays

- As presented: deletion at the end in $O(1)$ (decrement n)
- But this means we never free memory, even if the array shrinks significantly: excessive space complexity
- Also: the analysis of a combination of insertions and deletions by the potential method is invalid if $2n - cap < 0$
- In practice: reduce array capacity when size decreases sufficiently
- Avoids constantly creating/destroying arrays when adding/removing elements near the capacity

Deletion in Dynamic Arrays

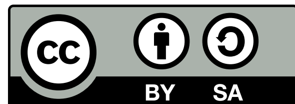
- As presented: deletion at the end in $O(1)$ (decrement n)
- But this means we never free memory, even if the array shrinks significantly: excessive space complexity
- Also: the analysis of a combination of insertions and deletions by the potential method is invalid if $2n - cap < 0$
- In practice: reduce array capacity when size decreases sufficiently
- Avoids constantly creating/destroying arrays when adding/removing elements near the capacity
- Full analysis: Chap. 17 of Cormen et al. [2009]

Insertion in a std::vector in C++



Licensing

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



Media used:

- Program memory layout diagram slide 19 is CC-BY-SA 3.0 by Dougct

Bibliography I

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.