Module 3

Memory management fundamentals

Memory management

- we can manage memory **manually** or **automatically** 😎

Memory management

- we can manage memory **manually** or **automatically** 😎
- if manually:

Memory management

- we can manage memory **manually** or **automatically** 😎
- if manually:
    - we create objects but we need also to delete them

Memory management

- we can manage memory **manually** or **automatically** 😎
- if manually:
    - we create objects but we need also to delete them
    - typically, creation is supported by some **allocator** - it's *"give me a new memory"* rather than *"give me **this** memory region"*

Memory management

- we can manage memory **manually** or **automatically** 😎
- if manually:
    - we create objects but we need also to delete them
    - typically, creation is supported by some **allocator** - it's *"give me a new memory"* rather than *"give me **this** memory region"*
- if automatically:

Memory management

- we can manage memory **manually** or **automatically** 😎
- if manually:
  - we create objects but we need also to delete them
  - typically, creation is supported by some **allocator** - it's *"give me a new memory"* rather than *"give me **this** memory region"*
- if automatically:
  - we only create objects and don't need to delete them - they will be automatically deleted

Memory management

- we can manage memory **manually** or **automatically** 😎
- if manually:
  - we create objects but we need also to delete them
  - typically, creation is supported by some **allocator** - it's *"give me a new memory"* rather than *"give me **this** memory region"*
- if automatically:
  - we only create objects and don't need to delete them - they will be automatically deleted
  - may be handled at:
    - **execution** time - both creation and reclamation are supported by **allocator** and some **garbage collection** mechanism
    - **compilation time** - discovering what should be reclaimed is discovered during compilation time

Memory management

- we can manage memory **manually** or **automatically** 😎
- if manually:
  - we create objects but we need also to delete them
  - typically, creation is supported by some **allocator** - it's *"give me a new memory"* rather than *"give me **this** memory region"*
- if automatically:
  - we only create objects and don't need to delete them - they will be automatically deleted
  - may be handled at:
    - **execution** time - both creation and reclamation are supported by **allocator** and some **garbage collection** mechanism
    - **compilation time** - discovering what should be reclaimed is discovered during compilation time (which is not trivial, we will see it soon!)

Manual memory management

Also known as *explicit allocation/deallocation*:

```c
#include<stdio.h>
int main()
{
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  if (ptr == 0)
  {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  *ptr = 25;
  printf("%d\n", *ptr);
  free(ptr);
  return 0;
}
```

Problems:

Manual memory management

Also known as *explicit allocation/deallocation*:

```c
#include<stdio.h>
int main()
{
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  if (ptr == 0)
  {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  *ptr = 25;
  printf("%d\n", *ptr);
  free(ptr);
  return 0;
}
```

Problems:

- tiresome - a lot of ceremony code, instead of business code

Manual memory management

Also known as *explicit allocation/deallocation*:

```
#include<stdio.h>
int main()
{
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  if (ptr == 0)
  {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  *ptr = 25;
  printf("%d\n", *ptr);
  free(ptr);
  return 0;
}
```

Problems:

- tiresome - a lot of ceremony code, instead of business code
- memory leak - if we forget to call `free` (😱)

Manual memory management

Also known as *explicit allocation/deallocation*:

```
#include<stdio.h>
int main()
{
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  if (ptr == 0)
  {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  *ptr = 25;
  printf("%d\n", *ptr);
  free(ptr);
  return 0;
}
```

Problems:

- tiresome - a lot of ceremony code, instead of business code
- memory leak - if we forget to call `free` (😱)
- dangling pointer - if we use a pointer after calling `free` (😱😱)

Manual memory management

Also known as *explicit allocation/deallocation*:

```c
#include<stdio.h>
int main()
{
  int *ptr;
  ptr = (int*)malloc(sizeof(int));
  if (ptr == 0)
  {
    printf("ERROR: Out of memory\n");
    return 1;
  }
  *ptr = 25;
  printf("%d\n", *ptr);
  free(ptr);
  return 0;
}
```

Problems:

- tiresome - a lot of ceremony code, instead of business code
- memory leak - if we forget to call `free` (😱)
- dangling pointer - if we use a pointer after calling `free` (😱😱)
- cross-library support - every library promotes it own "helpers" to handle that (aka *smart pointers*)

Automatic memory management

Addresses common manual memory management problems:

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
  - in other words, regular code should not seen a reference **after** the memory has been reclaimed

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
  - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
    - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
    - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

- less or more overhead - "automagic" have some costs:

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
    - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

- less or more overhead - "automagic" have some costs:
    - in case of *execution time* - we need some **runtime**/**garbage collection** process, and it will just consume resources and influence (pause? slowdown?) the program

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
  - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

- less or more overhead - "automagic" have some costs:
  - in case of *execution time* - we need some **runtime**/**garbage collection** process, and it will just consume resources and influence (pause? slowdown?) the program
  - in case of *compile time* - smart enough compiler will be just slower (and probably still need some *ceremony code* from the developer)

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
  - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

- less or more overhead - "automagic" have some costs:
  - in case of *execution time* - we need some **runtime**/**garbage collection** process, and it will just consume resources and influence (pause? slowdown?) the program
  - in case of *compile time* - smart enough compiler will be just slower (and probably still need some *ceremony code* from the developer)
- it is an abstraction of infinite memory:

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
  - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

- less or more overhead - "automagic" have some costs:
  - in case of *execution time* - we need some **runtime**/**garbage collection** process, and it will just consume resources and influence (pause? slowdown?) the program
  - in case of *compile time* - smart enough compiler will be just slower (and probably still need some *ceremony code* from the developer)
- it is an abstraction of infinite memory:
  - sometimes it "leaks" - implementation details (sometimes complex and unclear) may become problematic in edge cases and non trivial to analyze

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
  - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

- less or more overhead - "automagic" have some costs:
  - in case of *execution time* - we need some **runtime**/**garbage collection** process, and it will just consume resources and influence (pause? slowdown?) the program
  - in case of *compile time* - smart enough compiler will be just slower (and probably still need some *ceremony code* from the developer)
- it is an abstraction of infinite memory:
  - sometimes it "leaks" - implementation details (sometimes complex and unclear) may become problematic in edge cases and non trivial to analyze
  - it is impossible to produce solution good enough for all possible scenarios - we need to start thinking how to choose/configure it

Automatic memory management

Addresses common manual memory management problems:

- tiresome - only business code, no ceremony
- memory leak - should not happen, *"`free` calls"* are somehow automagically injected
- dangling pointer - pointers/references are managed and it should not happen
  - in other words, regular code should not seen a reference **after** the memory has been reclaimed
- cross-library support - *lingua franca* of the whole ecosystem

But... adds some new problems:

- less or more overhead - "automagic" have some costs:
  - in case of *execution time* - we need some **runtime**/**garbage collection** process, and it will just consume resources and influence (pause? slowdown?) the program
  - in case of *compile time* - smart enough compiler will be just slower (and probably still need some *ceremony code* from the developer)
- it is an abstraction of infinite memory:
  - sometimes it "leaks" - implementation details (sometimes complex and unclear) may become problematic in edge cases and non trivial to analyze
  - it is impossible to produce solution good enough for all possible scenarios - we need to start thinking how to choose/configure it
- BTW, **that's why we are here in this course** 😅

Automatic memory management

Two groups of algorithms:

- reference counting (and similar)
- tracing garbage collector

Reference counting

Reference counting

- simple idea:

Reference counting

- simple idea:
    - automatically **counts** how many references/pointers point to an object

Reference counting

- simple idea:
  - automatically **counts** how many references/pointers point to an object
  - when the count drops to 0 - it's high time we removed an object

Reference counting

- simple idea:
  - automatically **counts** how many references/pointers point to an object
  - when the count drops to 0 - it's high time we removed an object
- implementations:

Reference counting

- simple idea:
  - automatically **counts** how many references/pointers point to an object
  - when the count drops to 0 - it's high time we removed an object
- implementations:
  - *smart pointers* without the runtime - C++, COM, Rust:

```
int main()
{
      std::shared_ptr<Foo> shared(new Foo);
      std::cout << shared.use_count() << '\n';
}
```

Reference counting

- simple idea:
    - automatically **counts** how many references/pointers point to an object
    - when the count drops to 0 - it's high time we removed an object
- implementations:
    - *smart pointers* without the runtime - C++, COM, Rust:

```
int main()
{
	std::shared_ptr<Foo> shared(new Foo);
	std::cout << shared.use_count() << '\n';
}
```

    - *smart pointers* with the runtime - Python

Reference counting

- advantages:

Reference counting

- advantages:
  - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**

Reference counting

- advantages:
    - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**
    - incremental work - we clean up **constantly**, no big "churns" of work

Reference counting

- advantages:
  - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**
  - incremental work - we clean up **constantly**, no big "churns" of work
  - simple implementation - at least until not doing something more sophisticated:

Reference counting

- advantages:
    - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**
    - incremental work - we clean up **constantly**, no big "churns" of work
    - simple implementation - at least until not doing something more sophisticated:
        - deferred reference counting - avoiding counting reference stored on the stack
        - RC Immix - Taking Off the Gloves with Reference Counting Immix paper, '2013

Reference counting

- advantages:
  - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**
  - incremental work - we clean up **constantly**, no big "churns" of work
  - simple implementation - at least until not doing something more sophisticated:
    - deferred reference counting - avoiding counting reference stored on the stack
    - RC Immix - Taking Off the Gloves with Reference Counting Immix paper, '2013
- disadvantages:

Reference counting

- advantages:
    - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**
    - incremental work - we clean up **constantly**, no big "churns" of work
    - simple implementation - at least until not doing something more sophisticated:
        - deferred reference counting - avoiding counting reference stored on the stack
        - RC Immix - Taking Off the Gloves with Reference Counting Immix paper, '2013
- disadvantages:
    - overhead - every assignment between references has a cost now

Reference counting

- advantages:
  - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**
  - incremental work - we clean up **constantly**, no big "churns" of work
  - simple implementation - at least until not doing something more sophisticated:
    - deferred reference counting - avoiding counting reference stored on the stack
    - RC Immix - Taking Off the Gloves with Reference Counting Immix paper, '2013
- disadvantages:
  - overhead - every assignment between references has a cost now

    (although *"recent work narrowed the gap between reference counting and the fastest tracing collectors to **within 10%**")*

Reference counting

- advantages:
  - deterministic deallocation - as soon we get 0, we can reclaim memory **immediately**
  - incremental work - we clean up **constantly**, no big "churns" of work
  - simple implementation - at least until not doing something more sophisticated:
    - deferred reference counting - avoiding counting reference stored on the stack
    - RC Immix - Taking Off the Gloves with Reference Counting Immix paper, '2013
- disadvantages:
  - overhead - every assignment between references has a cost now

    (although *"recent work narrowed the gap between reference counting and the fastest tracing collectors to **within 10%**"*)
  - cyclic references - needs some additional help (overhead)

Reference counting on steroids

- *affine type system* - every variable (resource) is used **at most once**, which is enforced by the compiler

Reference counting on steroids

- *affine type system* - every variable (resource) is used **at most once**, which is enforced by the compiler
- the compiler limits the usage of resources - in a way that allows to track resources usage

Reference counting on steroids

- *affine type system* - every variable (resource) is used **at most once**, which is enforced by the compiler
- the compiler limits the usage of resources - in a way that allows to track resources usage
    - in other words, if there is **always** at most only one owner of data, we know we don't need it when the owner dies!

Reference counting on steroids

- *affine type system* - every variable (resource) is used **at most once**, which is enforced by the compiler
- the compiler limits the usage of resources - in a way that allows to track resources usage
  - in other words, if there is **always** at most only one owner of data, we know we don't need it when the owner dies!
- Rust is a new unmanaged language (without the runtime) with affine type system - thanks to techniques like **move semantics**, **ownership** and **borrowing**

Affine type system - Rust

Move semantics example:

```rust
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Affine type system - Rust

Move semantics example:

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:4:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
```

Affine type system - Rust

Move semantics example:

```rust
fn main() {
    let s = String::from("hello");  // s comes into scope
    takes_ownership(s);             // s value moves into the function...
                                    // ... and so is no longer valid here


}
fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.
```

Affine type system - Rust

Borrowing example:

```
fn main() {
    let s = String::from("hello");
    use_string(&s);
    // string goes of scope here
}

fn use_string(some_string: &String) {
    // string has been borrowed, increases it's "counter usage"
}
```

Affine type system - Rust

Borrowing example:

```rust
fn main() {
    let s = String::from("hello");
    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

Affine type system - Rust

Borrowing example:

```
fn main() {
    let s = String::from("hello");
    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

```
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind a `&` reference
 --> src/main.rs:8:5
  |
7 | fn use_string(some_string: &String) {
  |                            ------- help: consider changing this to be a mutable reference: `&mut String`
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ `some_string` is a `&` reference, so the data it refers to cannot be borrowed as mutable
```

Affine type system - Rust

Borrowing example:

```rust
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Affine type system - Rust

Borrowing example:

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

Affine type system - Rust

Borrowing example:

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:4:14
  |
3 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
4 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
5 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here
```

Affine type system - Rust

Borrowing example:

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

Affine type system - Rust

Borrowing example:

```rust
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:5:14
  |
3 |     let r1 = &s; // no problem
  |              -- immutable borrow occurs here
4 |     let r2 = &s; // no problem
5 |     let r3 = &mut s; // BIG PROBLEM
  |              ^^^^^^ mutable borrow occurs here
6 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here
```

Tracing garbage collector

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
  - how & when to trace?

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
  - how & when to trace?
  - what does *"object is used"* mean?

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
  - how & when to trace?
  - what does *"object is used"* mean?
  - how & when to reclaim?

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
  - how & when to trace?
  - what does *"object is used"* mean?
  - how & when to reclaim?
- we have many tracing GCs implementation in managed runtimes (.NET, JVM, Ruby, Go, ...)

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
    - how & when to trace?
    - what does *"object is used"* mean?
    - how & when to reclaim?
- we have many tracing GCs implementation in managed runtimes (.NET, JVM, Ruby, Go, ...)
- the concept is old:

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
    - how & when to trace?
    - what does *"object is used"* mean?
    - how & when to reclaim?
- we have many tracing GCs implementation in managed runtimes (.NET, JVM, Ruby, Go, ...)
- the concept is old:
    - John McCarthy. *Recursive functions of symbolic expressions and their computation by machine.* - 1958

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
  - how & when to trace?
  - what does *"object is used"* mean?
  - how & when to reclaim?
- we have many tracing GCs implementation in managed runtimes (.NET, JVM, Ruby, Go, ...)
- the concept is old:
  - John McCarthy. *Recursive functions of symbolic expressions and their computation by machine.* - 1958
  - *"**John McCarthy** - American computer scientist and cognitive scientist. McCarthy was **one of the founders of the discipline of artificial intelligence**. (...), **developed the Lisp programming language** (...) significantly **influenced the design of the ALGOL** programming language, (...) and **invented garbage collection**.*"

Tracing garbage collector

- "simple" idea - **trace** which objects are used, and **collect** (delete) those which are not (😇)
- many aspects of implementation:
  - how & when to trace?
  - what does *"object is used"* mean?
  - how & when to reclaim?
- we have many tracing GCs implementation in managed runtimes (.NET, JVM, Ruby, Go, ...)
- the concept is old:
  - John McCarthy. *Recursive functions of symbolic expressions and their computation by machine.* - 1958
  - *"**John McCarthy** - American computer scientist and cognitive scientist. McCarthy was **one of the founders of the discipline of artificial intelligence**. (...), **developed the Lisp programming language** (...) significantly **influenced the design of the ALGOL** programming language, (...) and **invented garbage collection**.*"
  - in other words... during **Lisp** language design

Tracing garbage collector

Tracing garbage collector

- advantages:

Tracing garbage collector

- advantages:
    - simple to use - no ceremony code (🥰)

Tracing garbage collector

- advantages:
  - simple to use - no ceremony code (🥰)
  - "faster" - no overhead during regular work, only occasional pauses

Tracing garbage collector

- advantages:
  - simple to use - no ceremony code (🥰)
  - "faster" - no overhead during regular work, only occasional pauses
  - handles cyclic references out of the box

Tracing garbage collector

- advantages:
  - simple to use - no ceremony code (🥰)
  - "faster" - no overhead during regular work, only occasional pauses
  - handles cyclic references out of the box
- disadvantages:

Tracing garbage collector

- advantages:
    - simple to use - no ceremony code (🥰)
    - "faster" - no overhead during regular work, only occasional pauses
    - handles cyclic references out of the box
- disadvantages:
    - complex - hard to implement, may leak, adds some overhead

Tracing garbage collector

- advantages:
  - simple to use - no ceremony code (🥰)
  - "faster" - no overhead during regular work, only occasional pauses
  - handles cyclic references out of the box
- disadvantages:
  - complex - hard to implement, may leak, adds some overhead
  - non-deterministic deallocation - most (all?) implementations reclaim memory some time **after** an object is no longer needed

Tracing garbage collector

- advantages:
  - simple to use - no ceremony code (🥰)
  - "faster" - no overhead during regular work, only occasional pauses
  - handles cyclic references out of the box
- disadvantages:
  - complex - hard to implement, may leak, adds some overhead
  - non-deterministic deallocation - most (all?) implementations reclaim memory some time **after** an object is no longer needed
  - not completely "pauseless"

```
THE GARBAGE COLLECTION HAS BEEN CALLED. SOME INTERESTING STATISTICS ARE AS FOLLOWS:
```

~1960 McCarthy team, MIT symposium

Tracing garbage collector

Conceptually consists of two simple(™) phases:

Tracing garbage collector

Conceptually consists of two simple(™) phases:

- **Mark phase** - to discover objects that may be deleted ("trace" them)

Tracing garbage collector

Conceptually consists of two simple(™) phases:

- **Mark phase** - to discover objects that may be deleted ("trace" them)
- **Collect phase** - to delete discovered objects

Tracing garbage collector

Conceptually consists of two simple(™) phases:

- **Mark phase** - to discover objects that may be deleted ("trace" them)
- **Collect phase** - to delete discovered objects
  - obviously, the main goal here is not to delete objects themselves, but to reclaim memory after them

Tracing garbage collector

Conceptually consists of two simple(™) phases:

- **Mark phase** - to discover objects that may be deleted ("trace" them)
- **Collect phase** - to delete discovered objects
  - obviously, the main goal here is not to delete objects themselves, but to reclaim memory after them
  - significant decision is: do we move objects (**Compact**) or not (**Sweep**)

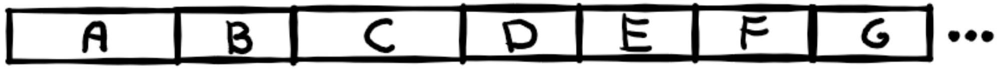
Tracing garbage collector

Conceptually consists of two simple(™) phases:

- **Mark phase** - to discover objects that may be deleted ("trace" them)
- **Collect phase** - to delete discovered objects
  - obviously, the main goal here is not to delete objects themselves, but to reclaim memory after them
  - significant decision is: do we move objects (**Compact**) or not (**Sweep**)
- both phases may be implemented "concurrently" (to the program execution and/or even to each other) and/or "incrementally"

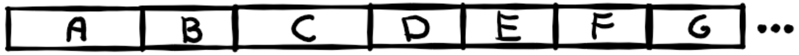
Tracing garbage collector - Mark

We need to know which objects are "used"...



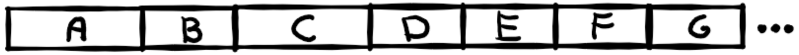
Tracing garbage collector - Mark & Object graph

In memory:



Tracing garbage collector - Mark & Object graph

In memory:

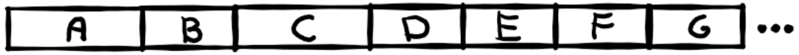


Type data:

```
record A(B b, D d);
record B(int X);
record C(B b, F f);
record D(E e);
record E(G g);
record F(int X);
record G(int Z);
```

Tracing garbage collector - Mark & Object graph

In memory:



Type data:

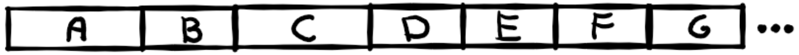
```
record A(B b, D d);
record B(int X);
record C(B b, F f);
record D(E e);
record E(G g);
record F(int X);
record G(int Z);
```

Current "state":

```
var a = new A(..., ...);
var d = new D(...);
...we are here...
```

Tracing garbage collector - Mark & Object graph

In memory:



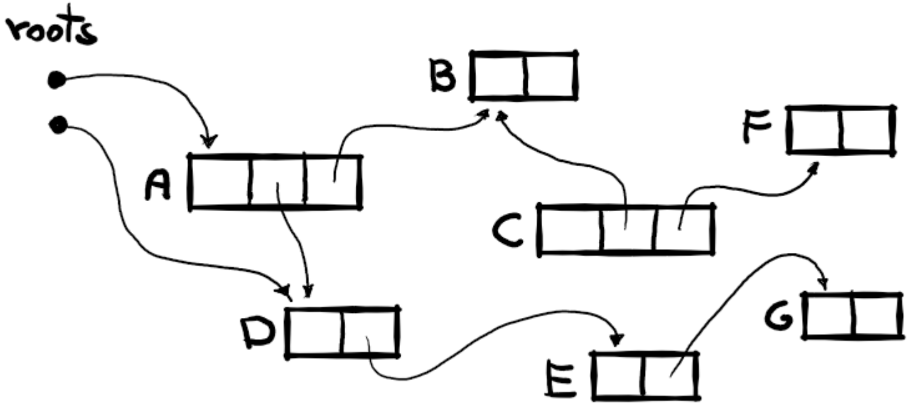
Type data:

```
record A(B b, D d);
record B(int X);
record C(B b, F f);
record D(E e);
record E(G g);
record F(int X);
record G(int Z);
```

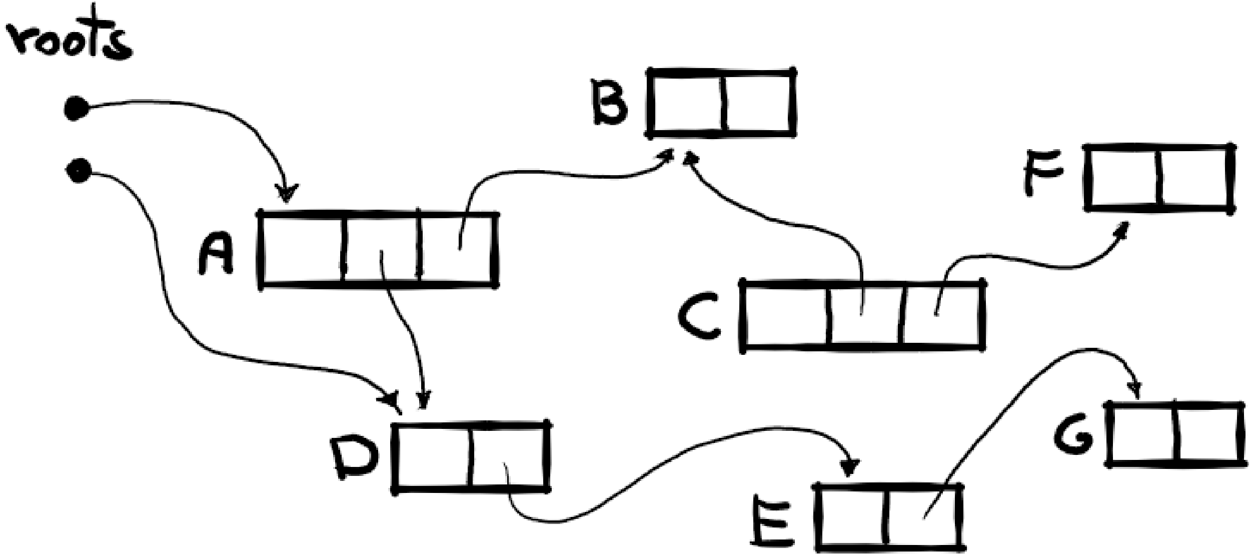
Current "state":

```
var a = new A(..., ...);
var d = new D(...);
...we are here...
```

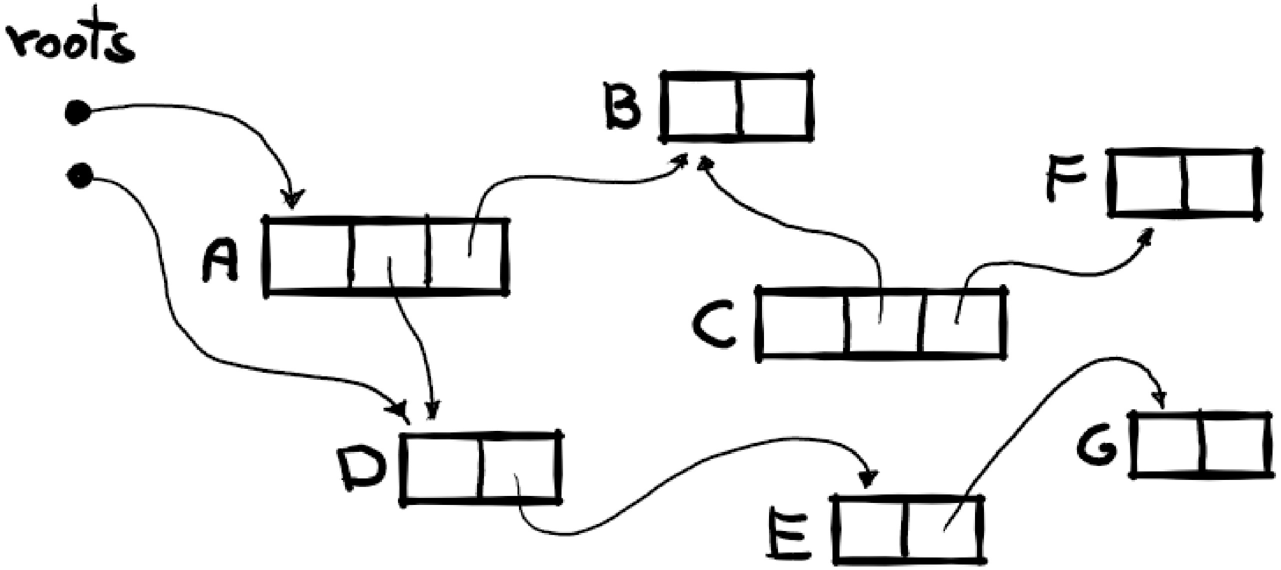
Object graph:



Object graph traversal

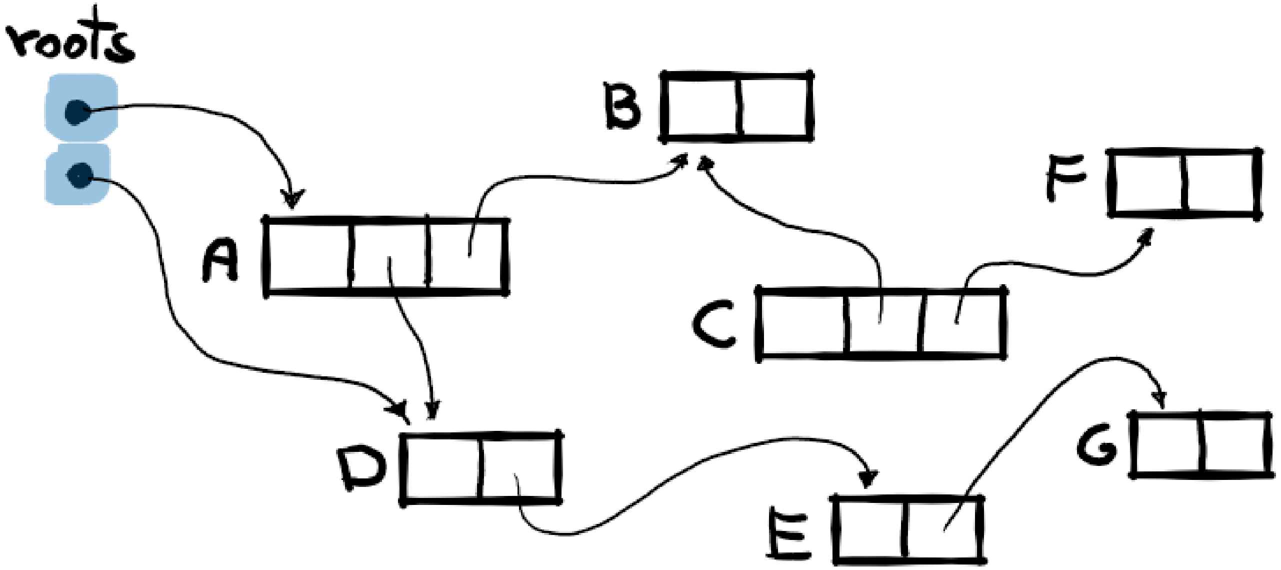


Object graph traversal



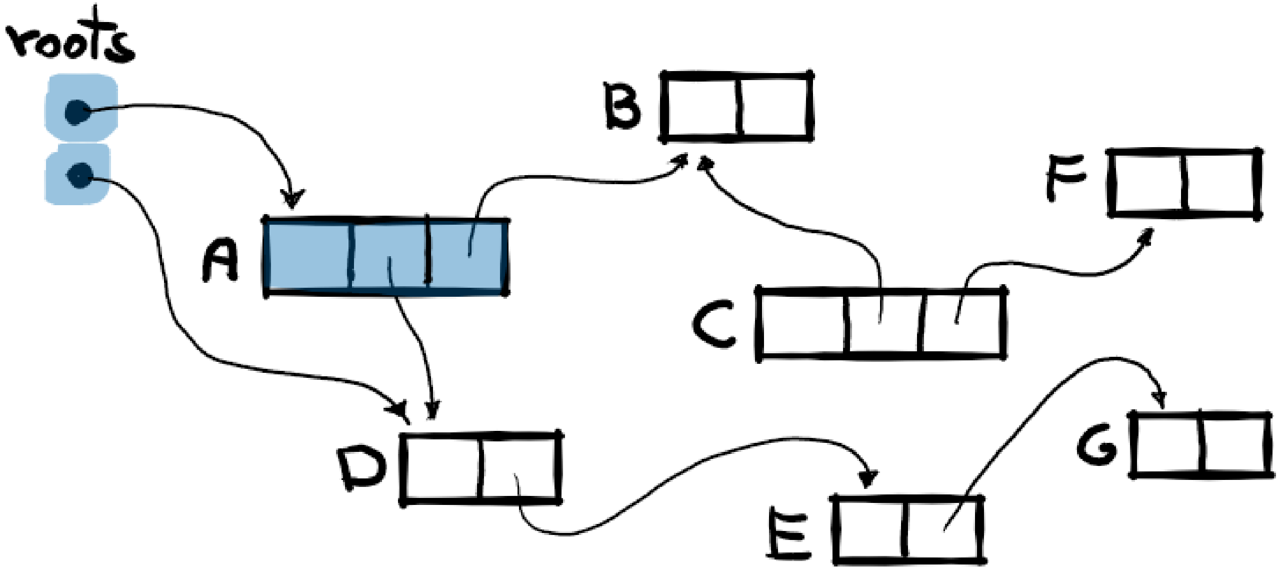
To visit:

Object graph traversal



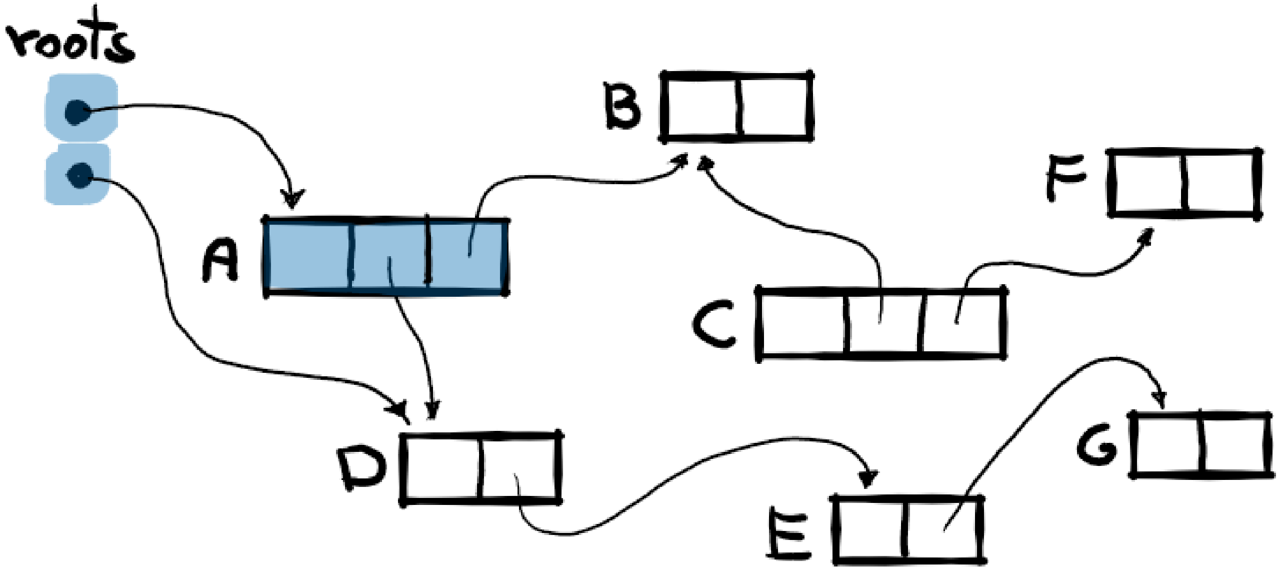
To visit: A, D

Object graph traversal



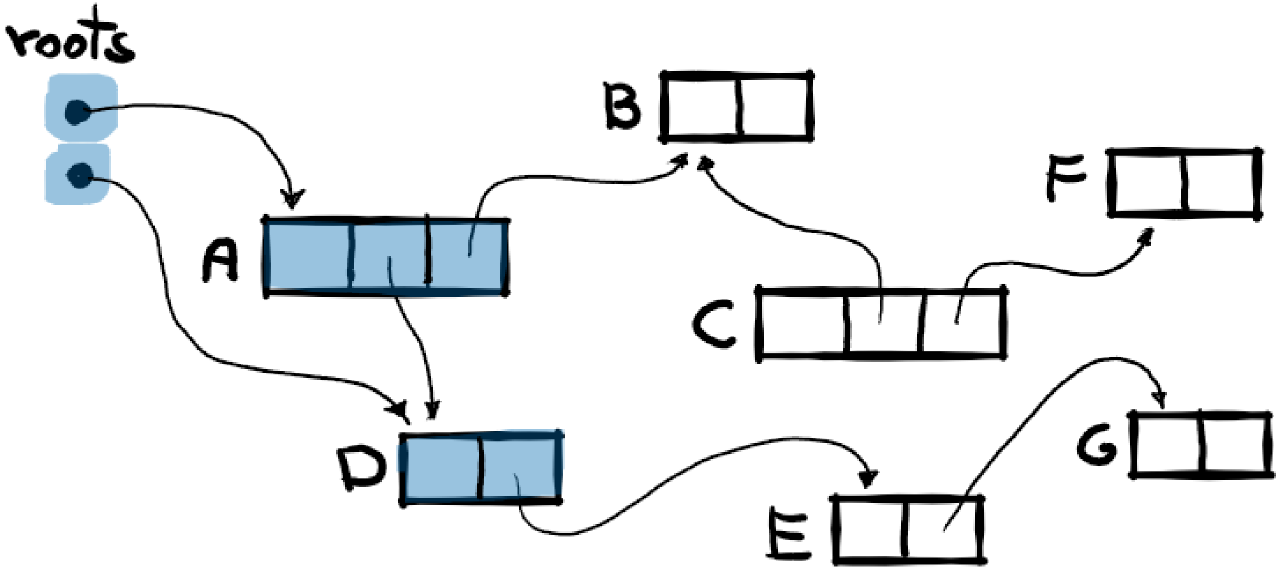
To visit: A, D, D, B

Object graph traversal



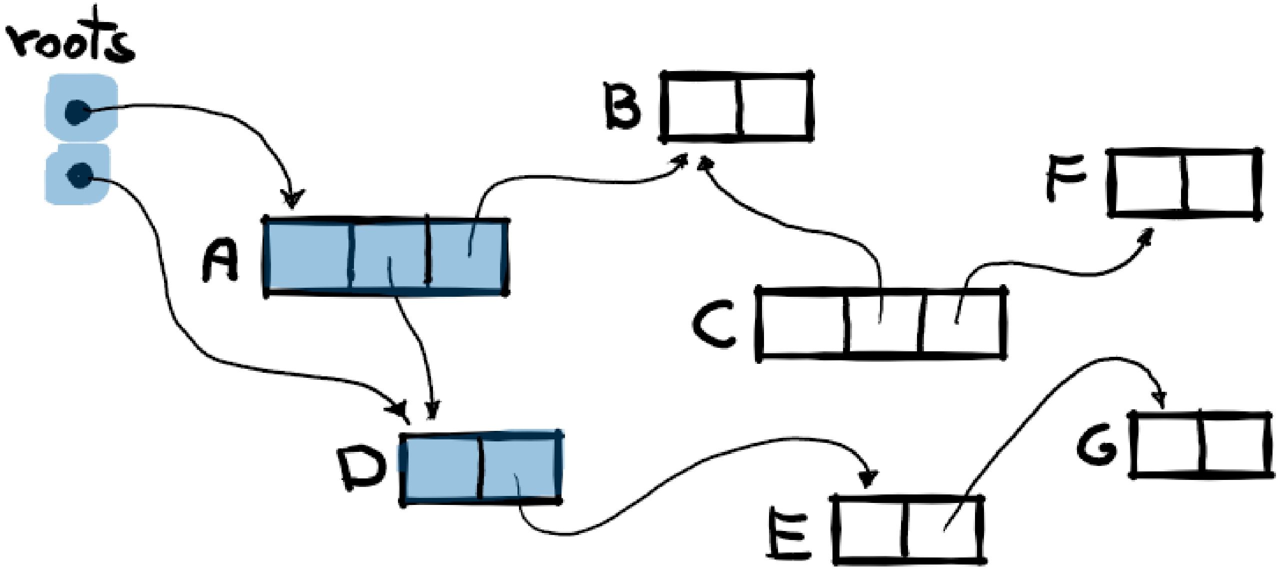
To visit: ~~A~~, D, D, B

Object graph traversal



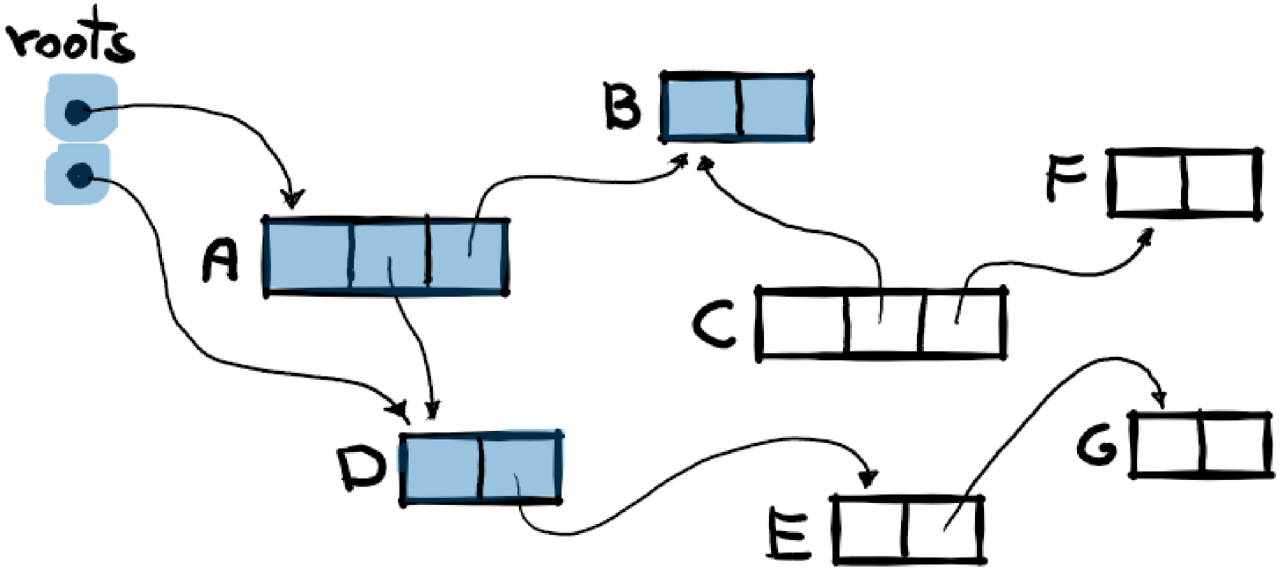
To visit: ~~A~~, D, D, B, E

Object graph traversal



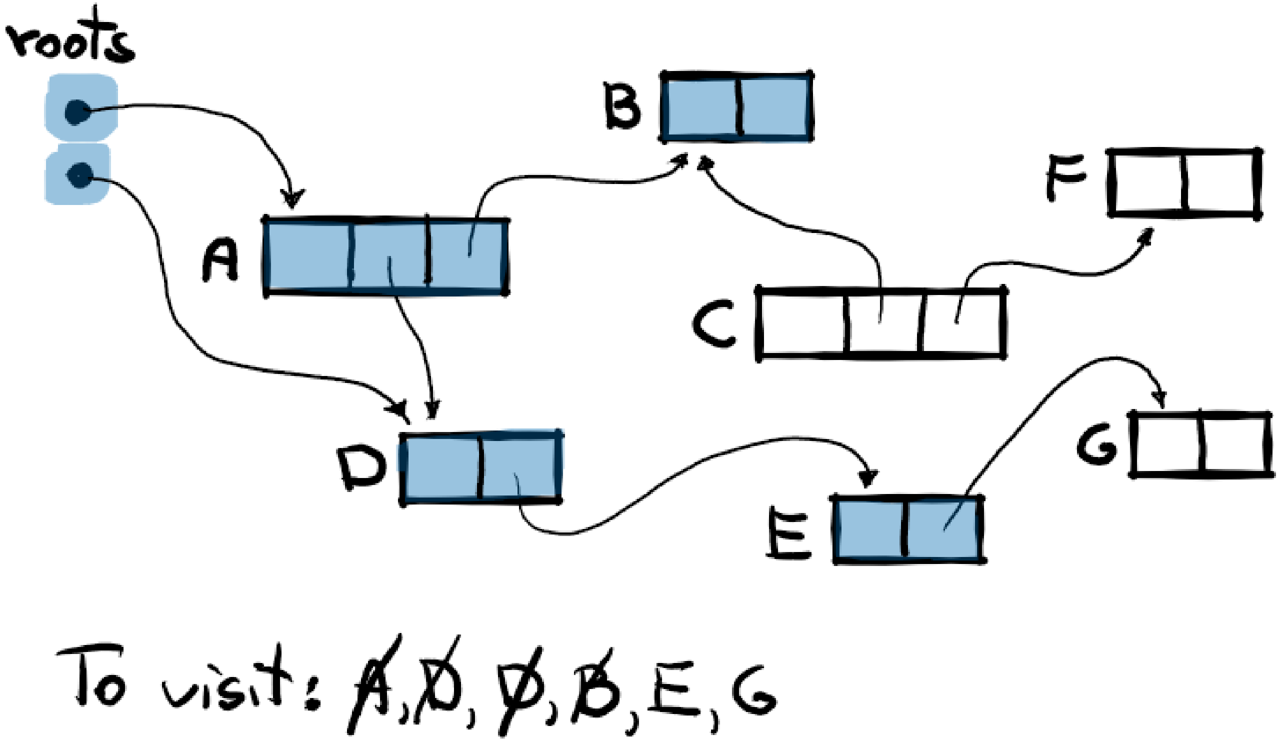
To visit: ~~A~~, ~~D~~, ~~D~~, B, E

Object graph traversal

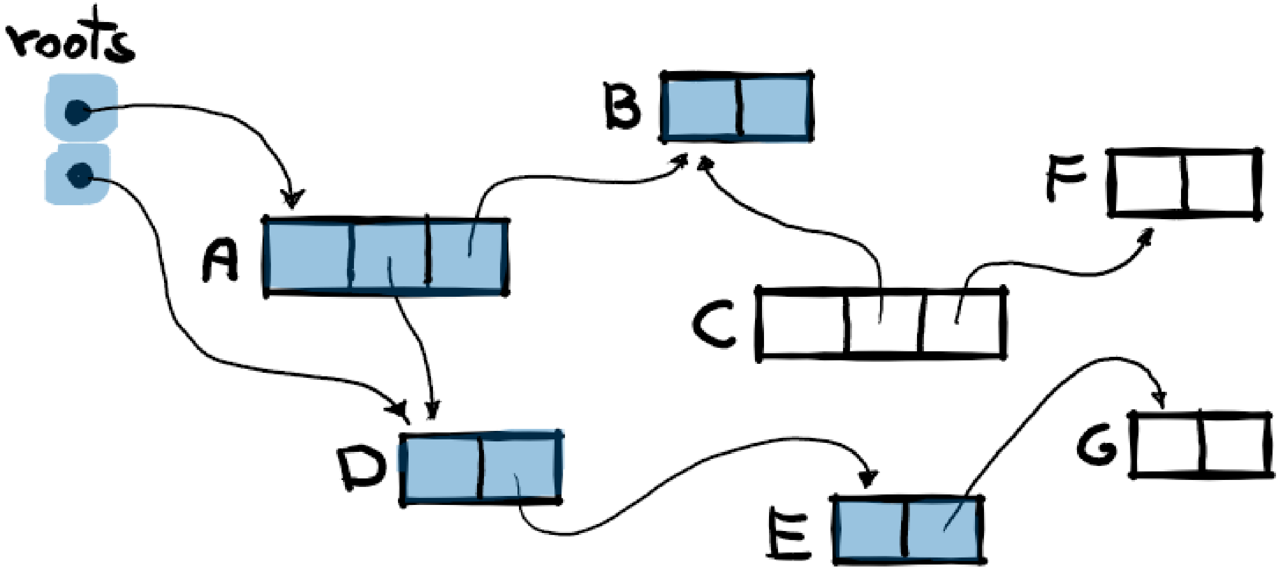


To visit: ~~A~~, ~~D~~, ~~D~~, B, E

Object graph traversal

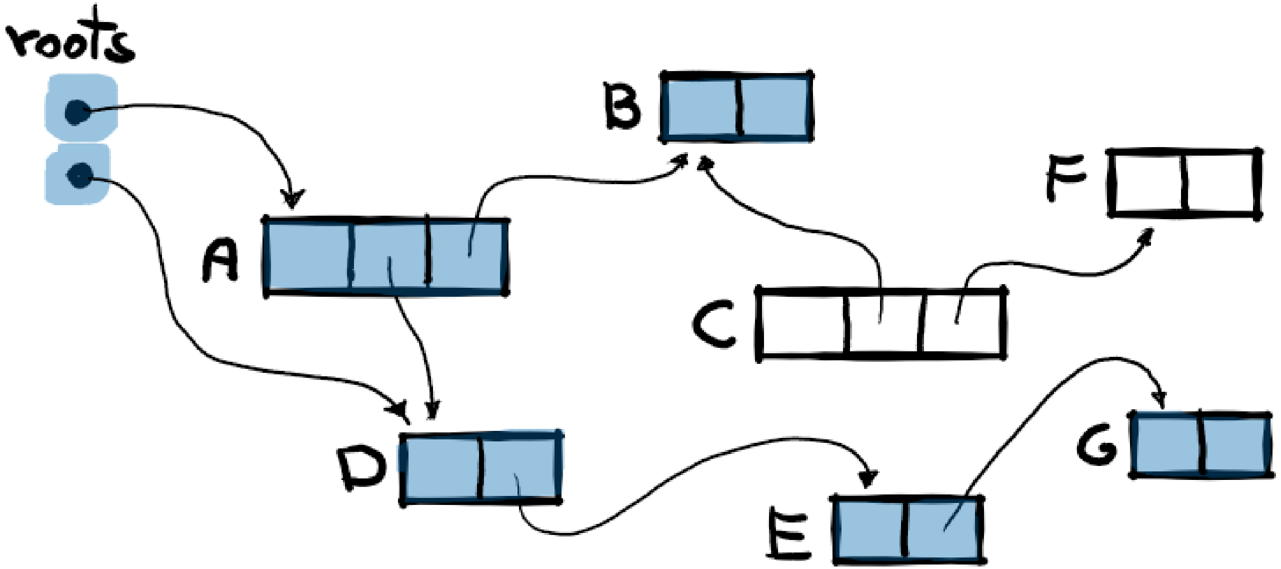


Object graph traversal



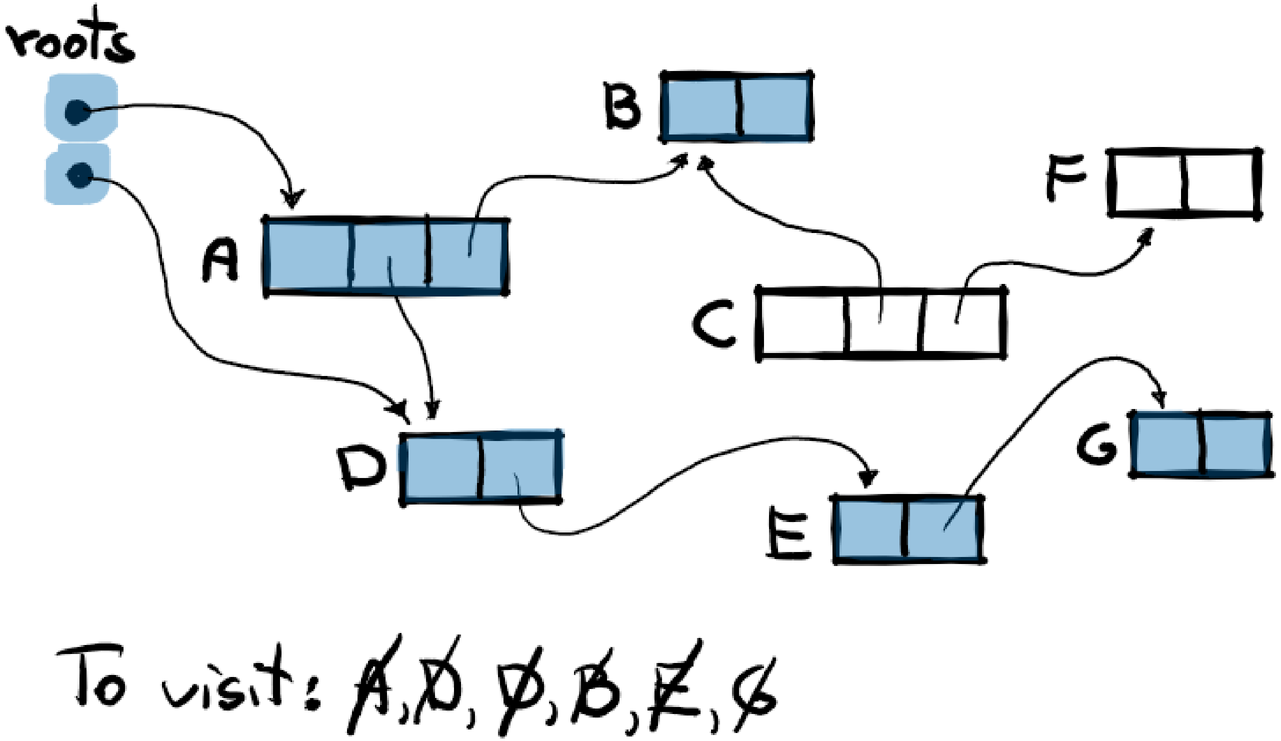
To visit: ~~A~~, ~~D~~, ~~D~~, ~~B~~, ~~E~~, G

Object graph traversal

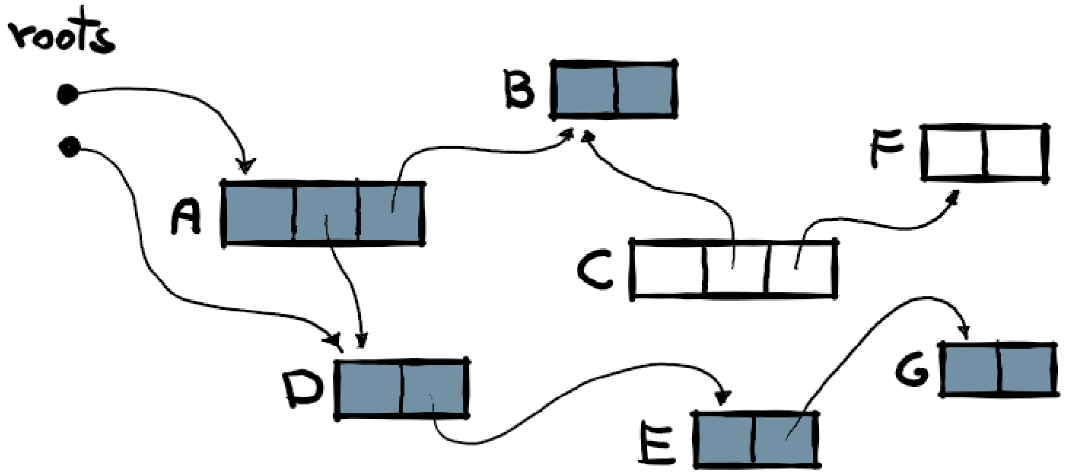


To visit: ~~A~~, ~~D~~, ~~D~~, ~~B~~, ~~E~~, G

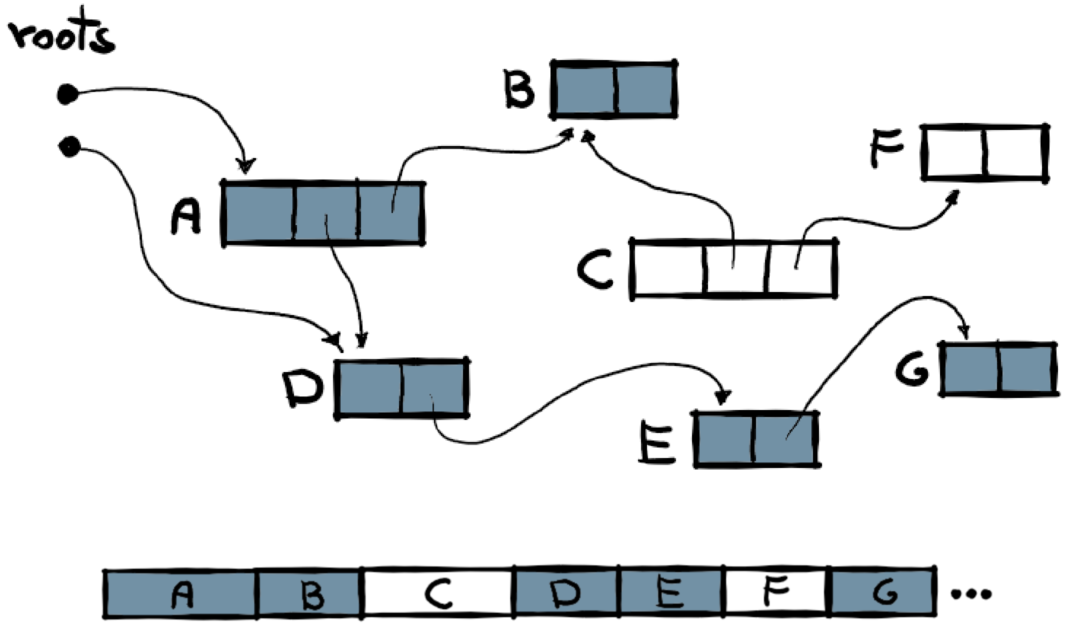
Object graph traversal



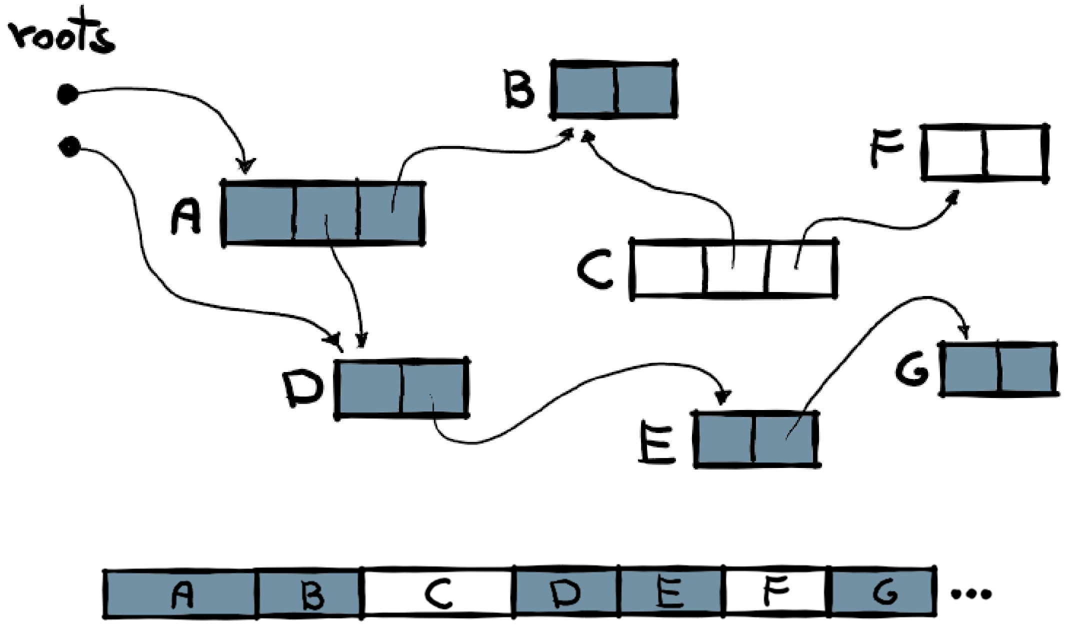
Object graph traversal



Object graph traversal

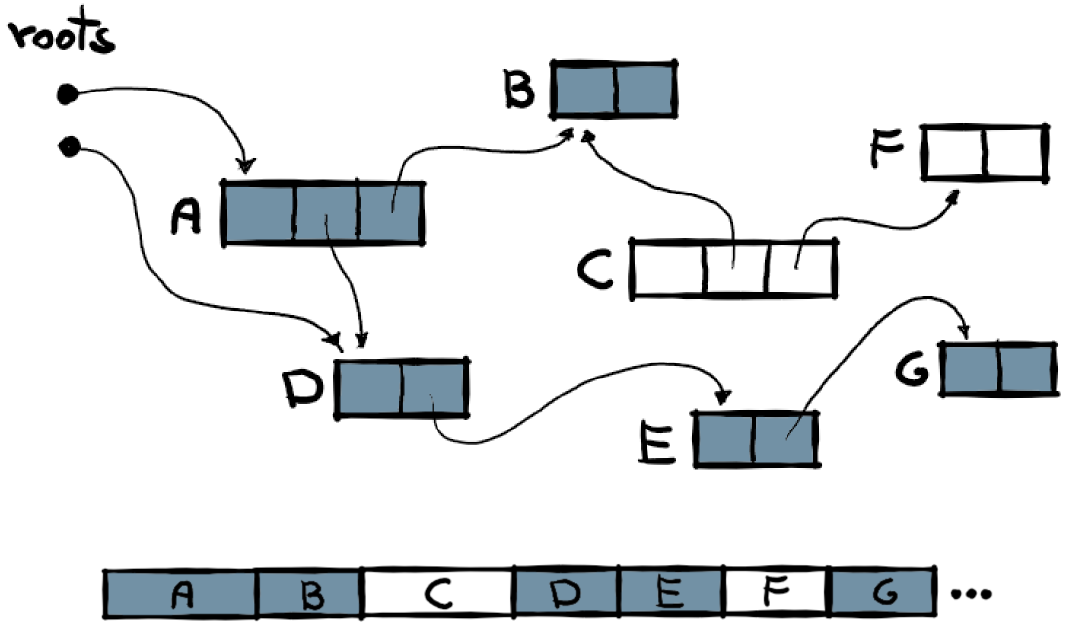


Object graph traversal



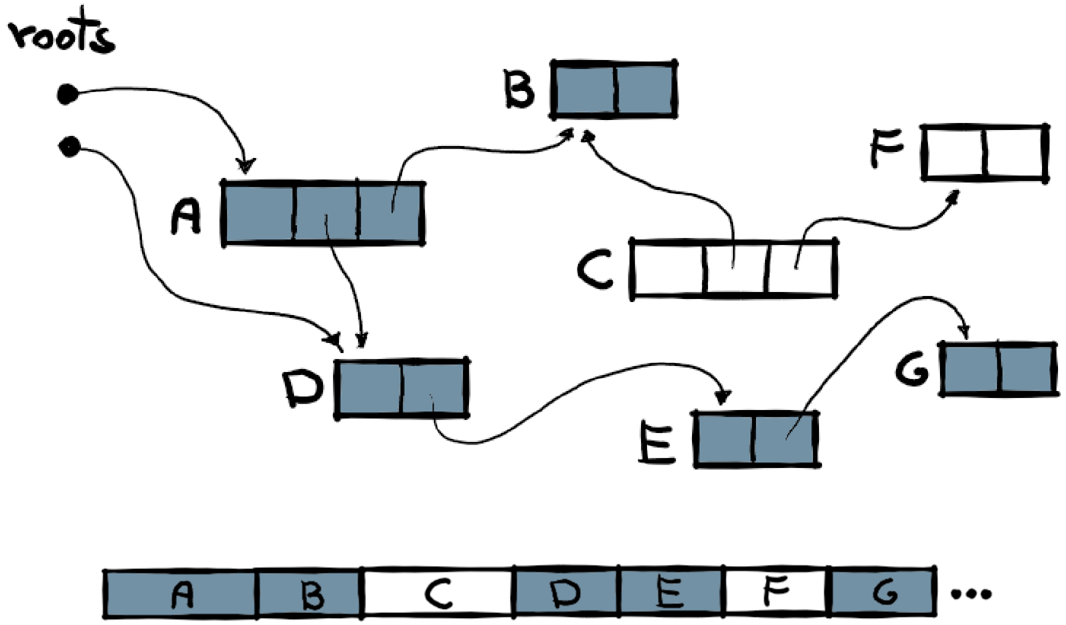
- we have just discovered **reachability** of the objects (from at least one root) by *marking* those reachable.

Object graph traversal



- we have just discovered **reachability** of the objects (from at least one root) by *marking* those reachable.
- **Reachability** is the closest we can get to true "usability" - we don't know the future.

Object graph traversal



- we have just discovered **reachability** of the objects (from at least one root) by *marking* those reachable.
- **Reachability** is the closest we can get to true "usability" - we don't know the future.
- objects `C` and `F` may now be deleted by the next phase

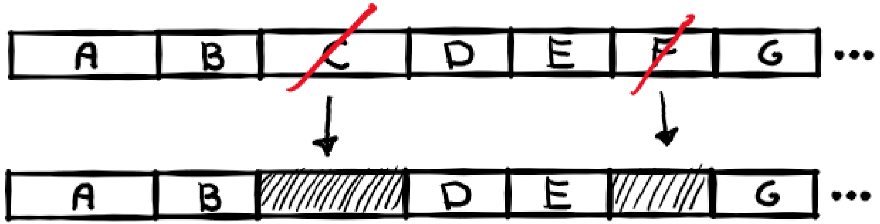
Object graph traversal

Possible roots:

- stack
- CPU registers
- static/thread-local static data
- finalization queue
- inter-generational references ("cards", "card tables")
  (we will return to that...)
- ...

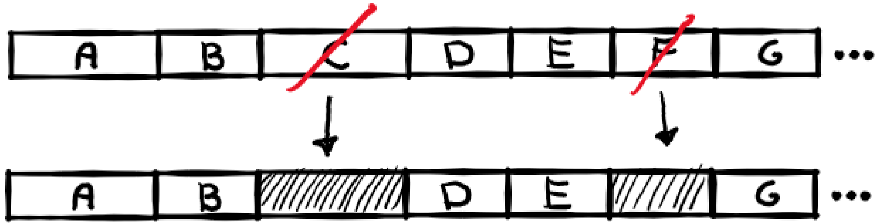
Collect - Sweep/Compact

Sweep



Collect - Sweep/Compact

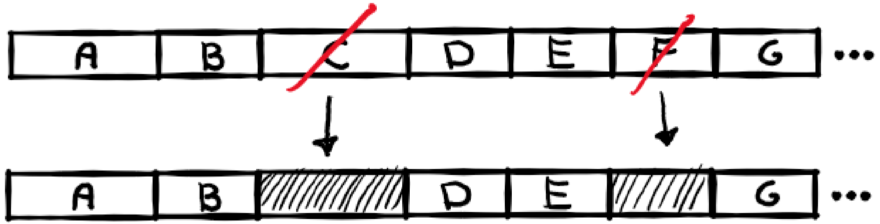
Sweep



- unreachable objects are treated as **free space**

Collect - Sweep/Compact

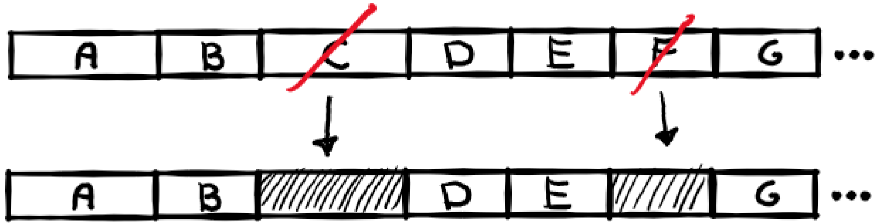
Sweep



- unreachable objects are treated as **free space**
    - most will be used as space for new objects

Collect - Sweep/Compact

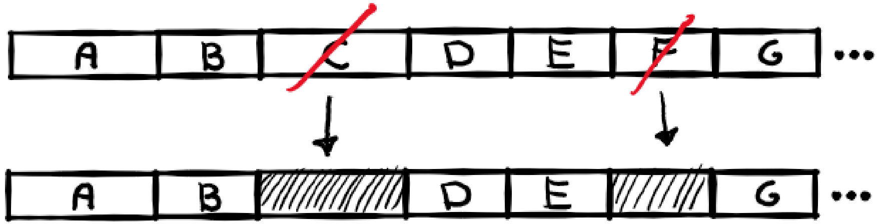
Sweep



- unreachable objects are treated as **free space**
    - most will be used as space for new objects
    - some will become unusable **fragmentation**

Collect - Sweep/Compact

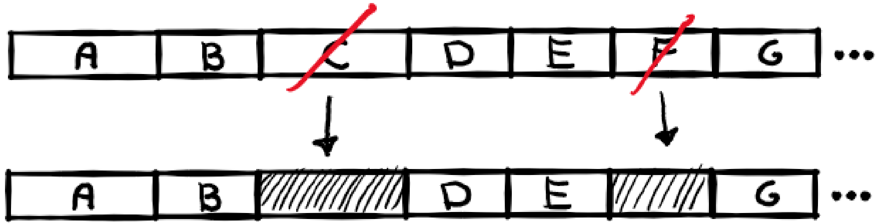
Sweep



- unreachable objects are treated as **free space**
  - most will be used as space for new objects
  - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)

Collect - Sweep/Compact

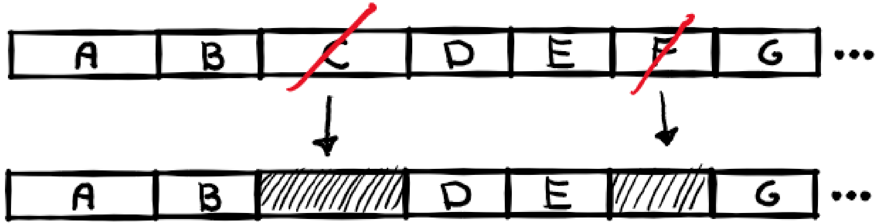
Sweep



- unreachable objects are treated as **free space**
    - most will be used as space for new objects
    - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)
- fast and simple operation (👍)

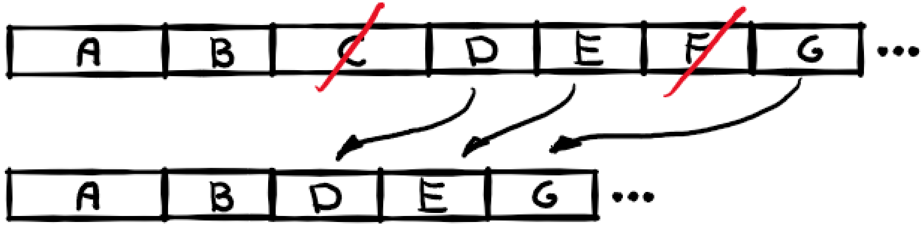
Collect - Sweep/Compact

Sweep



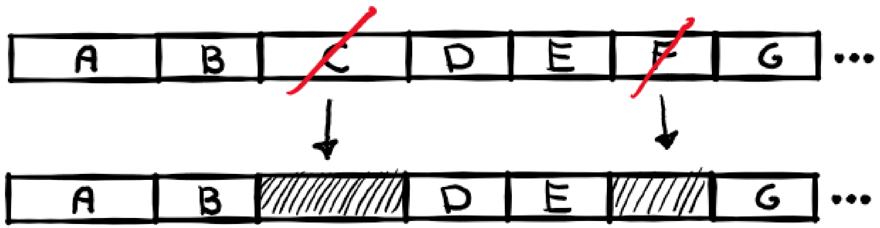
- unreachable objects are treated as **free space**
  - most will be used as space for new objects
  - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)
- fast and simple operation (👍)

Compact



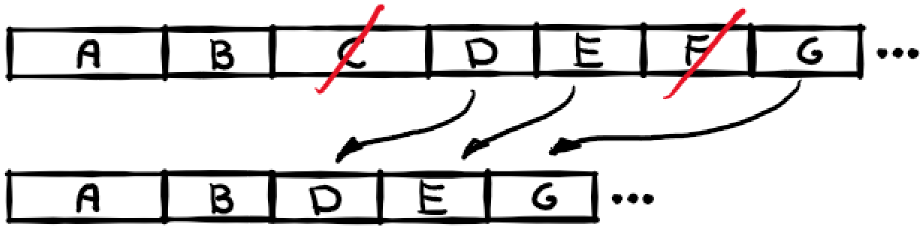
Collect - Sweep/Compact

Sweep



- unreachable objects are treated as **free space**
  - most will be used as space for new objects
  - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)
- fast and simple operation (👍)

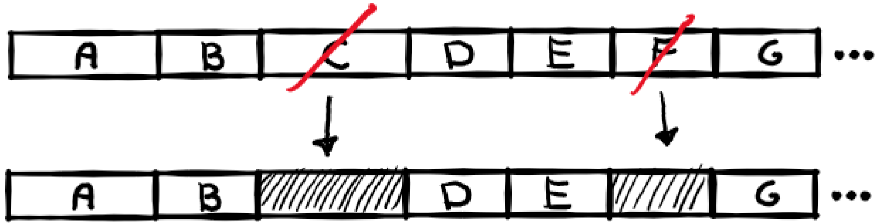
Compact



- move objects so gaps will disappear

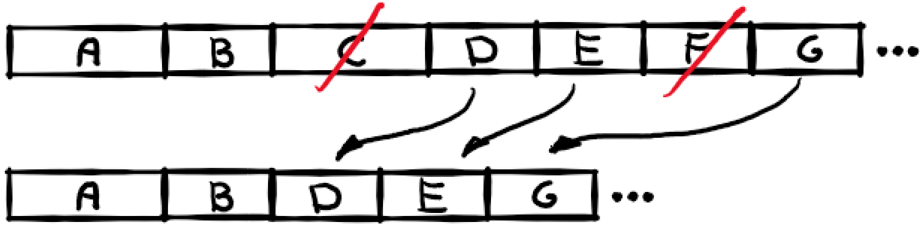
Collect - Sweep/Compact

Sweep



- unreachable objects are treated as **free space**
  - most will be used as space for new objects
  - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)
- fast and simple operation (👍)

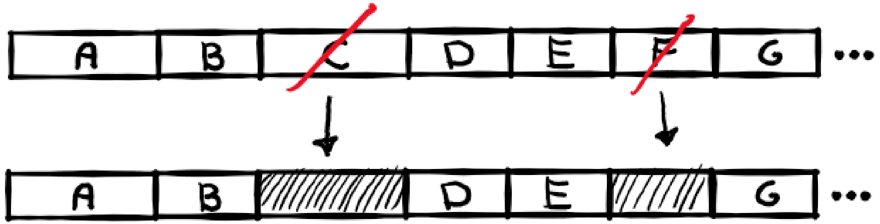
Compact



- move objects so gaps will disappear
- new space for objects is now at the end

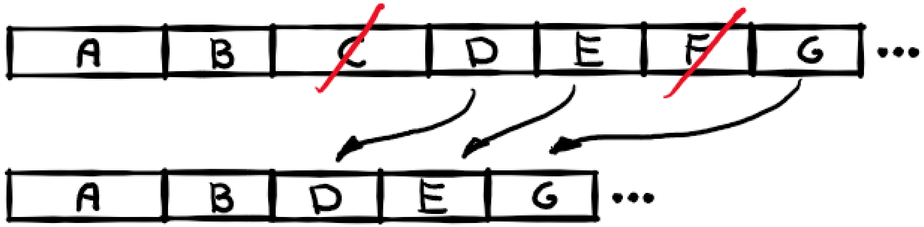
Collect - Sweep/Compact

Sweep



- unreachable objects are treated as **free space**
  - most will be used as space for new objects
  - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)
- fast and simple operation (👍)

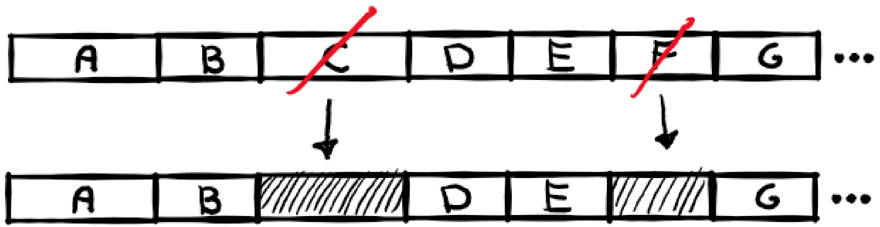
Compact



- move objects so gaps will disappear
- new space for objects is now at the end
- but... typically **we MAY reduce** memory usage - return this space to the operating system (👍)

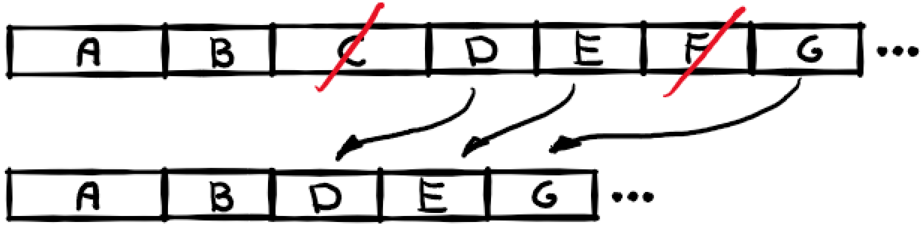
Collect - Sweep/Compact

Sweep



- unreachable objects are treated as **free space**
  - most will be used as space for new objects
  - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)
- fast and simple operation (👍)

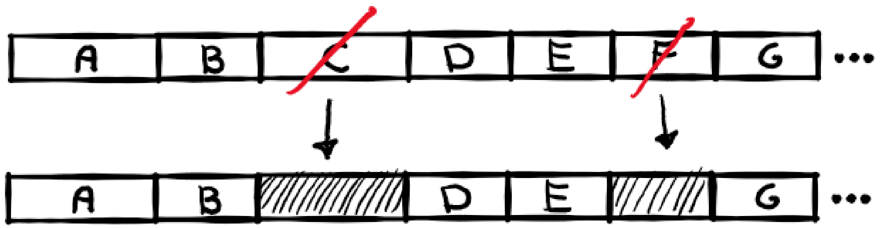
Compact



- move objects so gaps will disappear
- new space for objects is now at the end
- but... typically **we MAY reduce** memory usage - return this space to the operating system (👍)
- pretty complex operation with the memory copying overhead (👎)

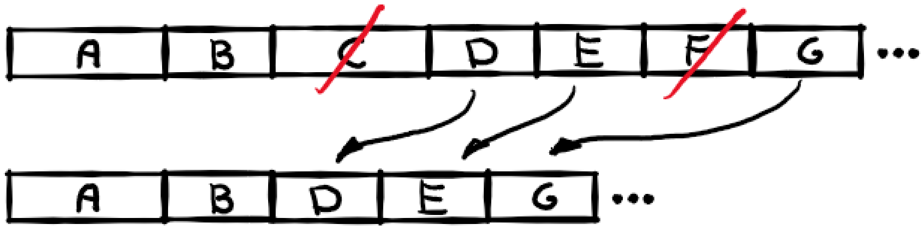
Collect - Sweep/Compact

Sweep



- unreachable objects are treated as **free space**
  - most will be used as space for new objects
  - some will become unusable **fragmentation**
- typically we **DON'T reduce** memory usage - free space is still assigned to the process (👎)
- fast and simple operation (👍)

Compact



- move objects so gaps will disappear
- new space for objects is now at the end
- but... typically **we MAY reduce** memory usage - return this space to the operating system (👍)
- pretty complex operation with the memory copying overhead (👎)

Question: how to make a decision whether to *sweep* or to *compact*...?! 🤔 We will return to that.

Automatic memory management

Two biggest lies:

Automatic memory management

Two biggest lies:

- there is **no** memory leak

Automatic memory management

Two biggest lies:

- there is **no** memory leak
- there is **no** memory leak unless we use unmanaged resources

Materials

- https://play.rust-lang.org
- Memory Management Reference
- The Garbage Collection Handbook: The Art of Automatic Memory Management book (💲)
- Fundamentals of garbage collection