Reference counting

Miroslaw Blazej
blazej@student.agh.edu.pl

Michal Dygas
dygas@student.agh.edu.pl

AGH University of Science and Technology

17-06-2020

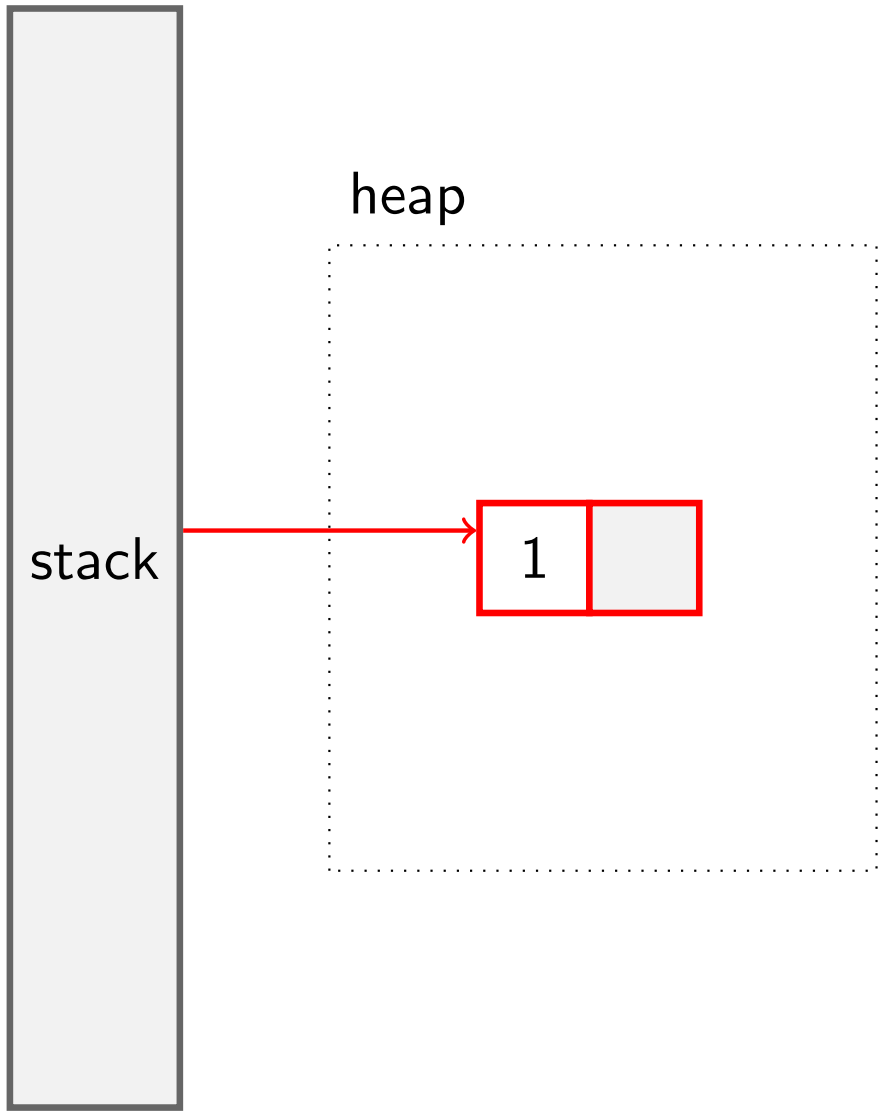
Introduction

Reference counting is an **garbage collection** algorithm. It is based on **recording the number of pointers that points to given chunk of memory**. When the number of pointers drops to zero, the chunk can be declared garbage.

In line with the name ‘reference counting‘, in this presentation pointers will be called ‘references’.

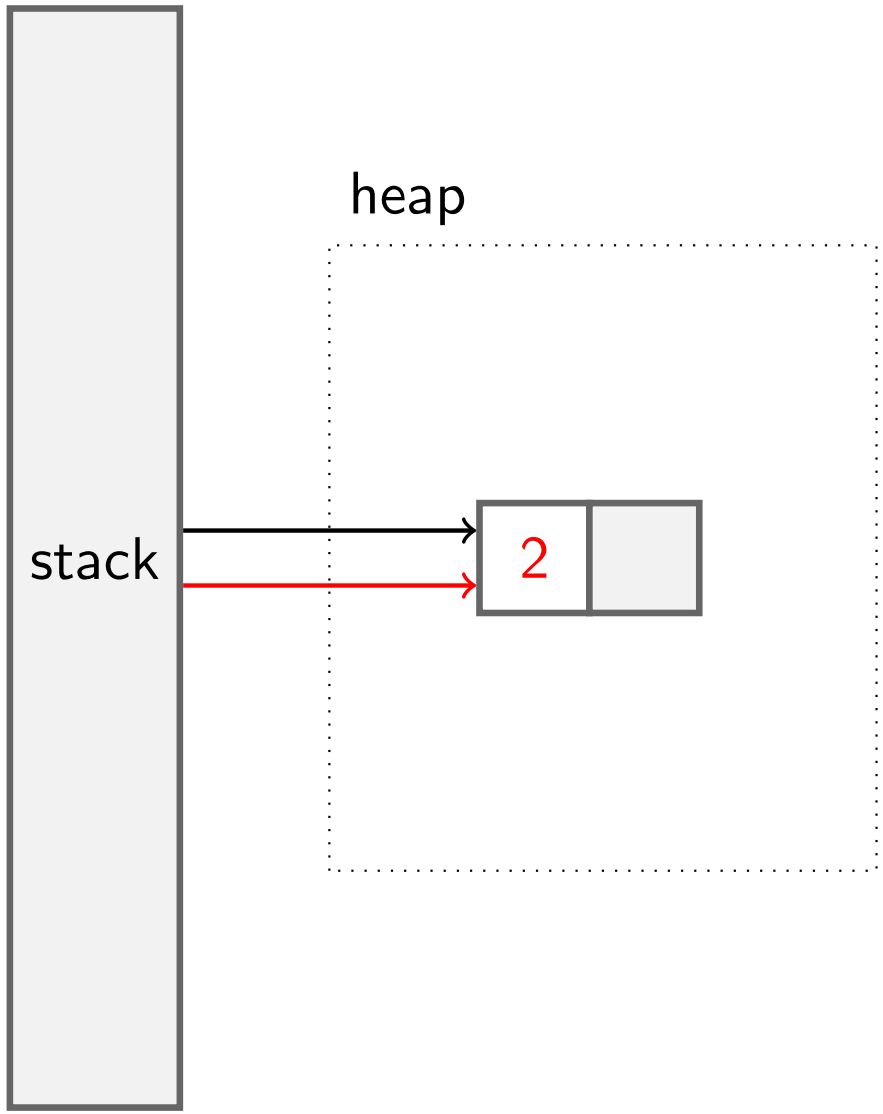
Basic rules (1)

1. When a chunk is allocated from the heap, its reference count is initialized to **one**.
2. Whenever a reference to the chunk is **duplicated**, its reference count is **increased by one** (incremented).
3. Whenever a reference to the chunk is **deleted**, its reference count is **decreased by one** (decremented).
4. If the reference count **drops to 0**, the chunk **can be freed** because it is no longer reachable



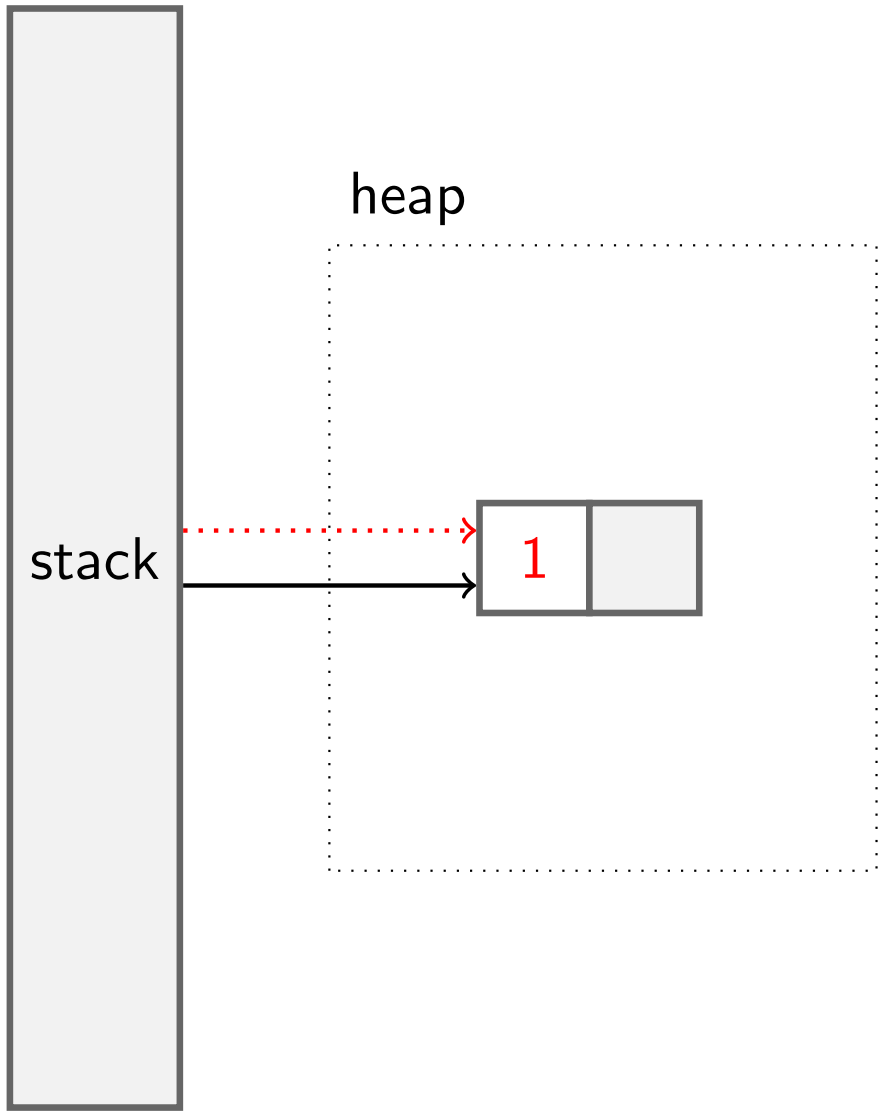
Basic rules (2)

1. When a chunk is allocated from the heap, its reference count is initialized to **one**.
2. Whenever a reference to the chunk is **duplicated**, its reference count is **increased by one** (incremented).
3. Whenever a reference to the chunk is **deleted**, its reference count is **decreased by one** (decremented).
4. If the reference count **drops to 0**, the chunk **can be freed** because it is no longer reachable



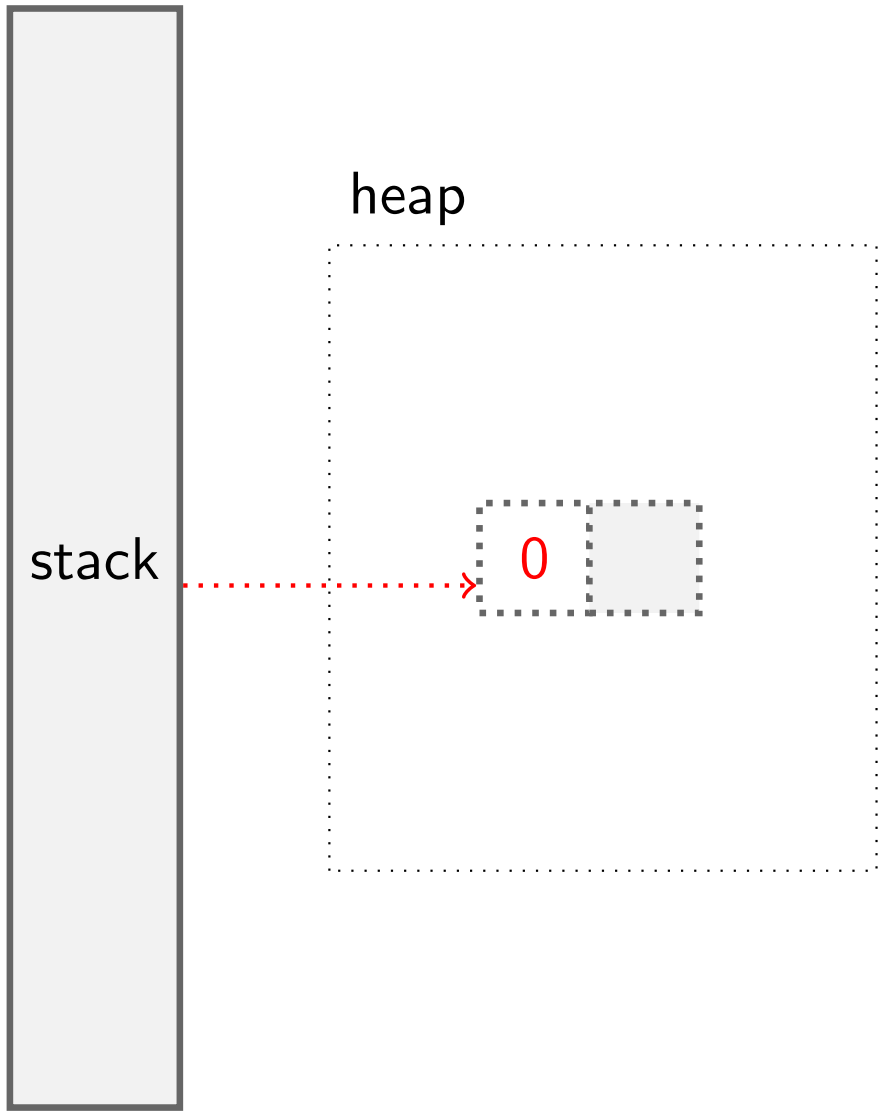
Basic rules (3)

1. When a chunk is allocated from the heap, its reference count is initialized to **one**.
2. Whenever a reference to the chunk is **duplicated**, its reference count is **increased by one** (incremented).
3. Whenever a reference to the chunk is **deleted**, its reference count is **decreased by one** (decremented).
4. If the reference count **drops to 0**, the chunk **can be freed** because it is no longer reachable



Basic rules (4)

1. When a chunk is allocated from the heap, its reference count is initialized to **one**.
2. Whenever a reference to the chunk is **duplicated**, its reference count is **increased by one** (incremented).
3. Whenever a reference to the chunk is **deleted**, its reference count is **decreased by one** (decremented).
4. If the reference count **drops to 0**, the chunk **can be freed** because it is no longer reachable



On-the-fly

Reference counting is **on-the-fly** type of garbage collector (also called incremental): some actions are performed at each call of `malloc` and/or `free`. These actions make some local modifications to the chunk structure to increase the probability of finding a free chunk when needed.

Main implementation challenges

1. keeping track of all **reference manipulations**
2. recursively freeing chunks with zero reference count

Reference manipulations

The compiler **inserts special code for all reference manipulations** and at each call of `malloc` and/or `free` - incrementing the reference count when a reference to a chunk is duplicated and decrementing it when such a reference is deleted.

Besides assignment statements, the compiler also has to add reference increasing code to **parameter transfers**, since a reference that is passed as a parameter is effectively assigned to a local variable of the called routine. Note that not all references in the running program are references to chunks on the heap; many of them point to blocks in the program data area, and all reference counting code must make sure it does not follow such references.

p := q additional reference manipulations pseudocode

```
if Points into the heap (q):
    Increment q.referenceCount;
if Points into the heap (p):
    Decrement p.referenceCount;
    if p.referenceCount = 0:
        FreeRecursively (p);
```

recursively freeing chunks pseudocode

```
procedure FreeRecursively(Pointer);
    if not IsPointerIntoHeap (Pointer): return;
    if Pointer.referenceCount != 0: return;
    for each i in 1 .. Pointer.numberOfPointers:
        if IsPointerIntoHeap (Pointer.pointer [i]):
            Decrement Pointer.pointer [i].referenceCount;
            FreeRecursively(Pointer.pointer [i]);
    FreeChunk(Pointer);
```

Recursively freeing cons

Cons

Recursion is, however, an unwelcome feature in a garbage collector since it requires an **unpredictable amount of stack space**.

Having a garbage collector fail for lack of memory is kind of embarrassing, though, and several techniques have been invented to avoid the problem. The best solution is using **pointer reversal**.

Identification techniques

figures that will illustrate this algorithm

Identification techniques

figures that will illustrate this algorithm

Algorithm issues

Cons

1. efficiency
2. memory fragmentation
3. failes to reclaim a cyclic data structure

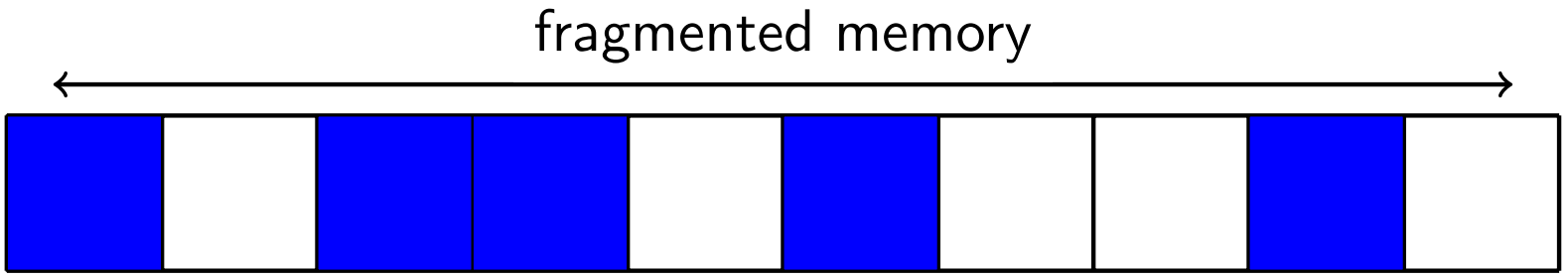
Efficiency

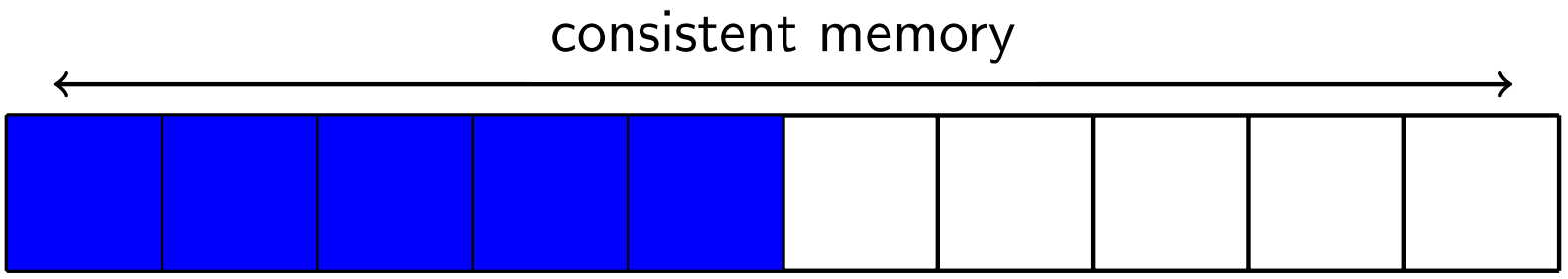
The compiled code has to monitor **all reference manipulations** and each and every reference manipulation requires the adjustment of the associated reference counts.

This is a considerable **overhead** in comparison to other garbage collection techniques that do not monitor any pointer action and reclaim garbage chunks only when needed.

Memory fragmentation

The free list is augmented with the reclaimed chunks, but it remains **fragmented**. In principle doing a compaction phase during a reference counting allocation request is possible, but few reference counting garbage collectors go to such lengths.





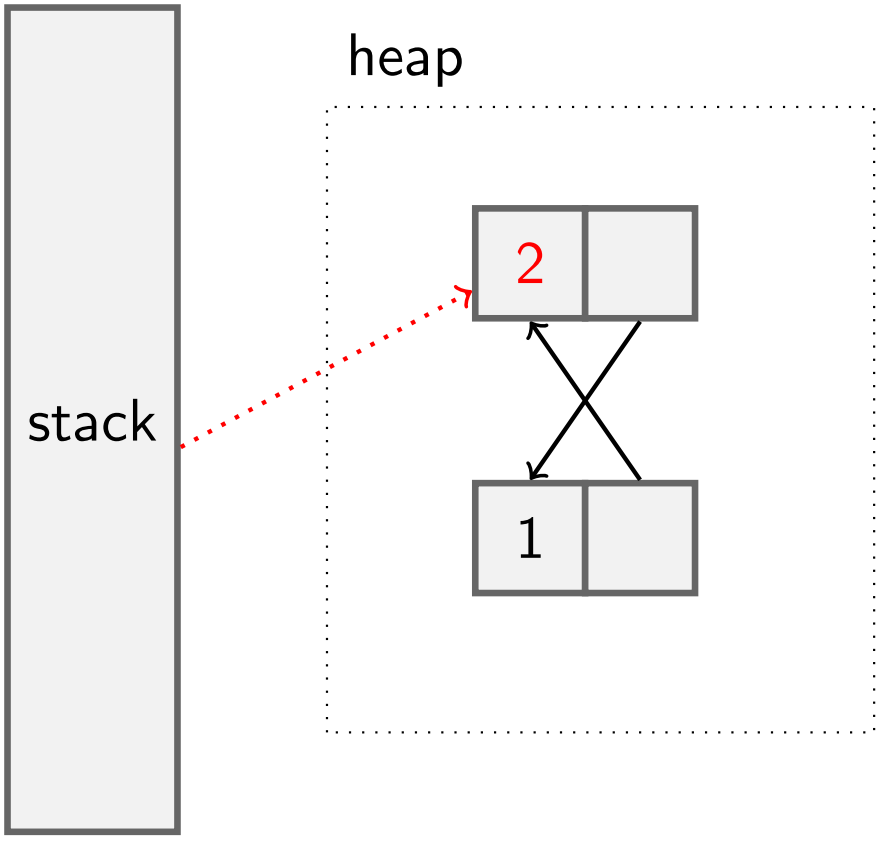
Cycles

The problem with reference counting is that it takes its decisions by considering **only one node** in the graph at a time and *in order to reclaim a cyclic data structure all nodes in the data structure should be considered as garbage together*.

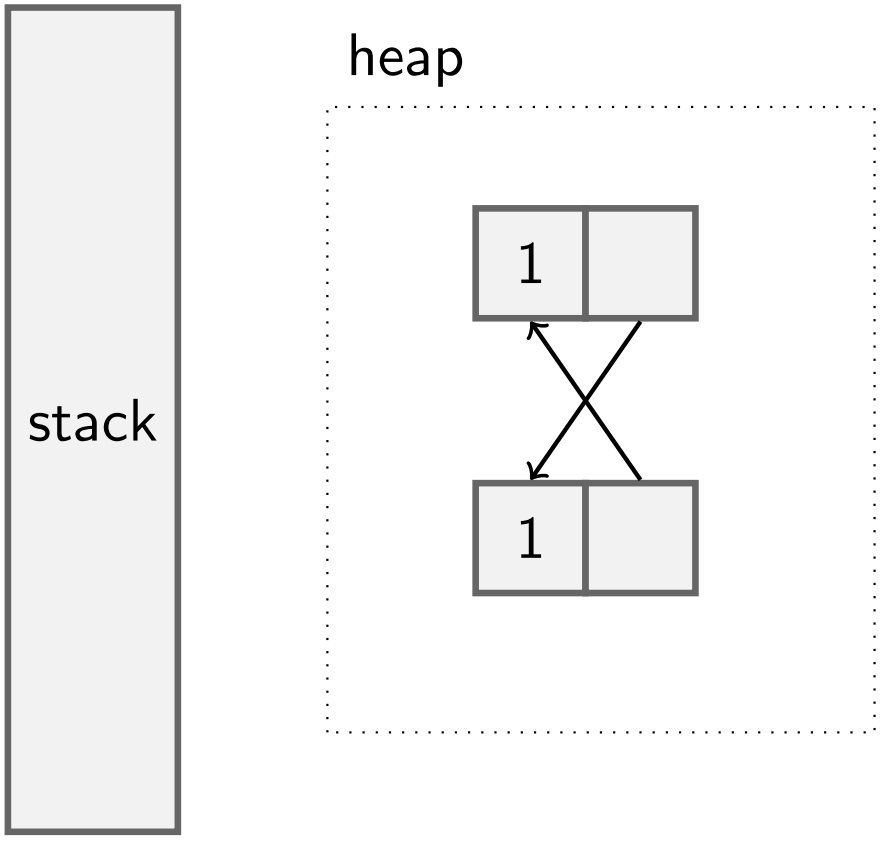
Once reference counting has failed to reclaim a cyclic data structure, the chunks involved **will never be reclaimed**. This has the unfortunate effect that free space leaks away, which might even cause the program to run out of free space when other garbage collectors would be able to reclaim the cyclic structures and allow the program to continue.

Cycles

Despite deletion of last reference to chunk from stack, which causes that chunk is no longer available...



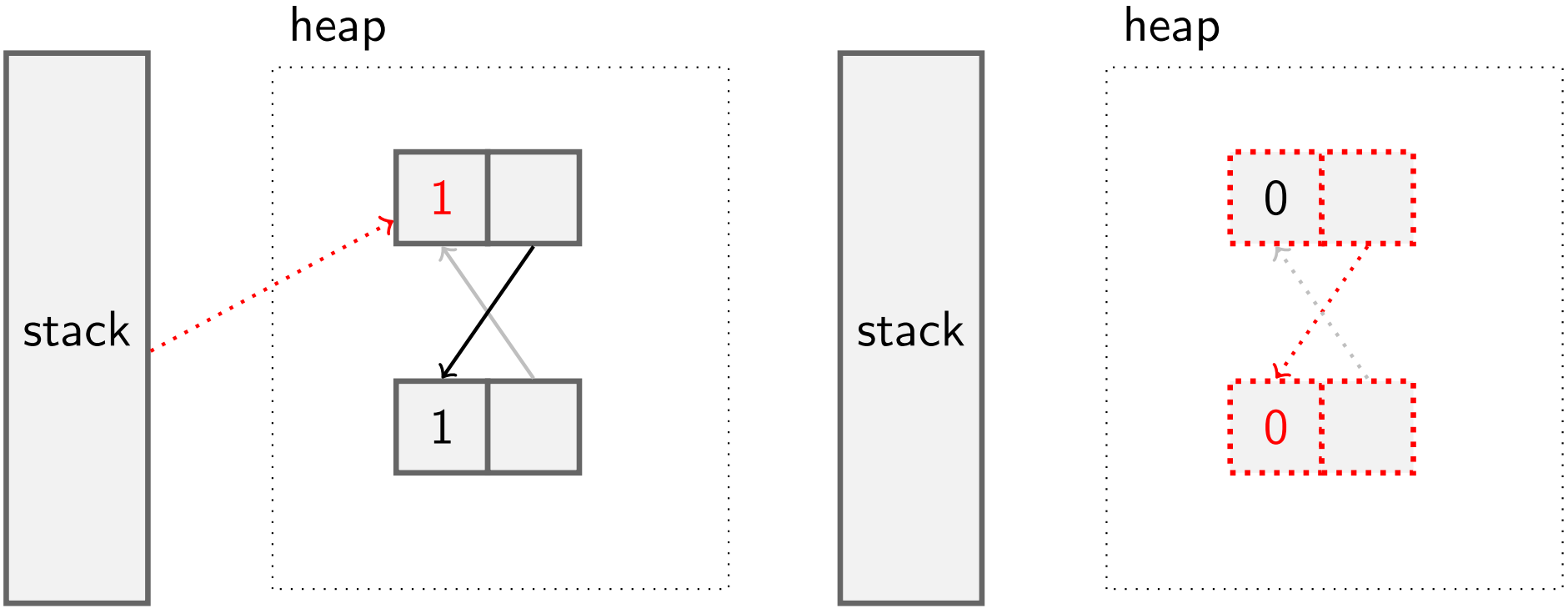
data is not deleted, as it is still referred by another chunk, kept alive by existing reference from our chunk



Weak references

When using reference counting, cycles can be implemented using **weak references**. Creating, manupulating and deleting them does not cause changes of reference count.

However, chunk referenced by weak reference might no longer exist; programmer has to manually handle such situation.



Reference counting in parallel programming

Reference counting is based on modyfing (incrementing or decrementing) a value every time reference is copied/dropped. If reference is shared between threads, more than one thread might try to copy/drop reference, causing **data race**.

This problem can be solved by making reference count an **atomic** variable. It solves problems with data races, hovewer - as atomic operations are expensive - it decerases performance.

Usage of reference counting - Objective-C, Swift

Reference counting is used as method of garbage collection in languages related to Apple ecosystem - **Objective-C** and **Swift**.

In order to allow cyclic data structures, Objective-C and Swift allows creating weak references using `weak` keyowrd:

```
class Person {
  var apartment: Apartment
}

class Apartment {
  weak var tenant: Person?
}
```

Usage of reference counting - smart pointers

Reference counting is also often used in **smart pointers**. Smart pointers are data structures that wraps raw pointers and manages data that it points to, providing garbage collection. They are **not a part of language itself** - they are usually part of standard library.

For example, C++ standard library provides `shared_ptr` and `weak_ptr` template classes, which implements reference counting:

```
shared_ptr<Person> person1 = make_shared<Person>(Person()) // refcount = 1
shared_ptr<Person> person2 = person // refcount = 2
weak_ptr<Person> person_weak = weak_ptr<Person>(person1) // refcount = 2
```

C++ mechanisms such as copying constructors, operator overloading and RAII allows smart pointers to use natural C++ idioms without need of directly implementing GC in the compiler.

Usage of reference counting - other

Rust also provides smart pointers types in its standard library. As avoiding data races in compile-time is main goal of Rust, Rust standard library provides 3 types of smart pointers:

- `Rc<T>` - does not use atomic variable for counting reference, cannot be sent to other threads (enforced in compile-time)
- `Arc<T>` - uses atomic variable for counting reference, can be sent to other threads
- `Weak<T>` - weak reference

Reference counting is also usually used in **UNIX** kernels for handling **file descriptiors recovery**.