Heaps and Garbage Collection

Madhavan Mukund, **S P Suresh**

Programming Language Concepts
Lecture 8, 02 February 2023

So far …

- **Static allocation** – for global variables

So far ...

- **Static allocation** – for global variables
    - Data stored in the data segment

So far ...

- **Static allocation** – for global variables
  - Data stored in the data segment
  - Referenced using offsets from a fixed location

So far ...

- **Static allocation** – for global variables
  - Data stored in the data segment
  - Referenced using offsets from a fixed location
- **Stack allocation** – for local variables, parameters etc.

So far ...

- **Static allocation** – for global variables
  - Data stored in the data segment
  - Referenced using offsets from a fixed location
- **Stack allocation** – for local variables, parameters etc.
  - Data stored in activation record of the function call

So far ...

- **Static allocation** – for global variables
  - Data stored in the data segment
  - Referenced using offsets from a fixed location
- **Stack allocation** – for local variables, parameters etc.
  - Data stored in activation record of the function call
  - Access to non-local variables by following a chain of **access links**

So far …

- **Static allocation** – for global variables
  - Data stored in the data segment
  - Referenced using offsets from a fixed location
- **Stack allocation** – for local variables, parameters etc.
  - Data stored in activation record of the function call
  - Access to non-local variables by following a chain of **access links**
  - **Control links** to go back to caller at the end of the function call

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x;  ...
    }
    public int f(int n) {
        int arr[n];  ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100);  ...
}
```

- Functions can handle complex data types – arrays / classes, ...

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x;  ...
    }
    public int f(int n) {
        int arr[n];  ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100);  ...
}
```

- Functions can handle complex data types – arrays / classes, ...
- **Dynamic data structures** like linked lists / graphs

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x;  ...
    }
    public int f(int n) {
        int arr[n];  ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100);  ...
}
```

- Functions can handle complex data types – arrays / classes, ...
- **Dynamic data structures** like linked lists / graphs
  - No pre-specified bound on the number of elements

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x;  ...
    }
    public int f(int n) {
        int arr[n];  ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100);  ...
}
```

- Functions can handle complex data types – arrays / classes, ...
- **Dynamic data structures** like linked lists / graphs
  - No pre-specified bound on the number of elements
- The activation record for `main` will store a **pointer** to the object `aObj` stored on the **heap**!

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x;  ...
    }
    public int f(int n) {
        int arr[n];  ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100);  ...
}
```

- Functions can handle complex data types – arrays / classes, …
- **Dynamic data structures** like linked lists / graphs
    - No pre-specified bound on the number of elements
- The activation record for `main` will store a **pointer** to the object `aObj` stored on the **heap**!
- `aObj` itself has pointers to the class definition

Dynamic allocation

```
class A {
    int x, y, z;
    A(x,y,z) {
        this.x = x;  ...
    }
    public int f(int n) {
        int arr[n];  ...
    }
}
main {
    A aObj(2,5,7);
    aObj.f(100);  ...
}
```

- Functions can handle complex data types – arrays / classes, …
- **Dynamic data structures** like linked lists / graphs
  - No pre-specified bound on the number of elements
- The activation record for `main` will store a **pointer** to the object `aObj` stored on the **heap**!
- `aObj` itself has pointers to the class definition
- The AR for `f` has a pointer to an array stored on heap

Heap

- **Heap** – just a chunk of memory

Heap

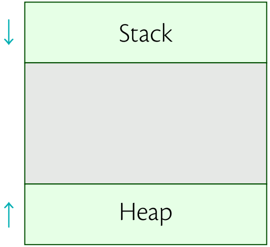
- **Heap** – just a chunk of memory
  - **Unstructured**

Heap

- **Heap** – just a chunk of memory
  - **Unstructured**
  - Nothing to do with the heap data structure used to implement priority queues!

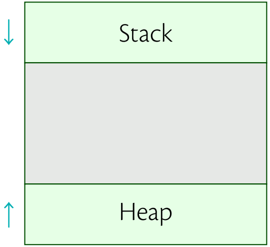
Heap

- **Heap** – just a chunk of memory
    - **Unstructured**
    - Nothing to do with the heap data structure used to implement priority queues!
- Typically depicted as "growing upward" (and the stack grows downward)



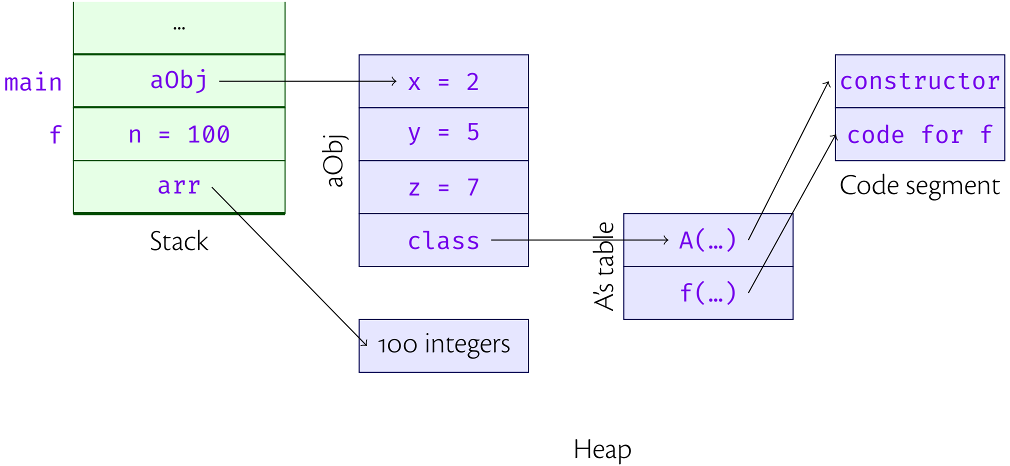
Heap

- **Heap** – just a chunk of memory
  - **Unstructured**
  - Nothing to do with the heap data structure used to implement priority queues!
- Typically depicted as "growing upward" (and the stack grows downward)



- Consist of chunks of **allocated** and **unallocated** memory

Stack and Heap



Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass

Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class

Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class
- Otherwise we look at the parent's table

Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class
- Otherwise we look at the parent's table
- Might need to follow a chain of pointers before we determine the code to run on a method call

Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class
- Otherwise we look at the parent's table
- Might need to follow a chain of pointers before we determine the code to run on a method call

- **Runtime polymorphism** has a simple implementation

Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class
- Otherwise we look at the parent's table
- Might need to follow a chain of pointers before we determine the code to run on a method call

- **Runtime polymorphism** has a simple implementation
- Consider an array of `Shape`s, each element being an instance of a subclass

Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class
- Otherwise we look at the parent's table
- Might need to follow a chain of pointers before we determine the code to run on a method call

- **Runtime polymorphism** has a simple implementation
- Consider an array of `Shape`s, each element being an instance of a subclass
- The elements of the arrays are pointers to objects

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class
- Otherwise we look at the parent’s table
- Might need to follow a chain of pointers before we determine the code to run on a method call

Overriding, inheritance etc.

- **Runtime polymorphism** has a simple implementation
- Consider an array of `Shape`s, each element being an instance of a subclass
- The elements of the arrays are pointers to objects
- The object data has a pointer to the precise subclass it is an instance of!

Overriding, inheritance etc.

- The table for each class has a pointer to the table for its superclass
- If we refer to a function which is overloaded, we find its pointer in the table for the class
- Otherwise we look at the parent's table
- Might need to follow a chain of pointers before we determine the code to run on a method call

- **Runtime polymorphism** has a simple implementation
- Consider an array of `Shape`s, each element being an instance of a subclass
- The elements of the arrays are pointers to objects
- The object data has a pointer to the precise subclass it is an instance of!
- Calling `perimeter` on each element of the array runs the code pointed to by the appropriate subclass table

Heaps and memory management

- As functions are called, they allocate data on the heap

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be **inaccessible** from stack!

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be **inaccessible** from stack!
- All computation and reference to data starts from the stack, but the data itself might be in heap

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be **inaccessible** from stack!
- All computation and reference to data starts from the stack, but the data itself might be in heap
- Allocated data might no longer have a reference from the stack (direct or indirect)

Heaps and memory management

- As functions are called, they allocate data on the heap
- At the end of the function, the allocated data on heap might not be needed
- Some data might be **inaccessible** from stack!
- All computation and reference to data starts from the stack, but the data itself might be in heap
- Allocated data might no longer have a reference from the stack (direct or indirect)
- This is called **garbage** – waste of memory

Explicit memory management

- Older languages expect programmer to manage memory

Explicit memory management

- Older languages expect programmer to manage memory
- `malloc / free` in C

Explicit memory management

- Older languages expect programmer to manage memory
- **`malloc / free`** in C
- **`new / delete`** in C++

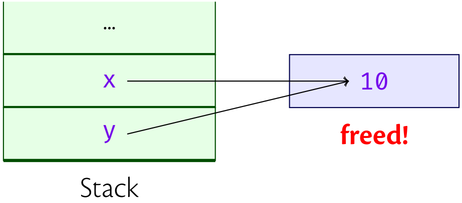
Explicit memory management

- Older languages expect programmer to manage memory
- **`malloc / free`** in C
- **`new / delete`** in C++
- **`free / delete`** tells the system to take back ownership of memory locations from the program – **deallocation**

Explicit memory management

- Older languages expect programmer to manage memory
- **malloc / free** in C
- **new / delete** in C++
- **free / delete** tells the system to take back ownership of memory locations from the program – **deallocation**
- Can cause the problem of **dangling pointers** – pointers to deallocated variables

```
int *x = malloc(
    sizeof(int));
*x = 10;
y = x;
free(x);
```



Garbage

- Dangling pointers are a serious problem!

Garbage

- Dangling pointers are a serious problem!
- Accessing a deallocated location could give arbitrary results

Garbage

- Dangling pointers are a serious problem!
- Accessing a deallocated location could give arbitrary results
- **Huge security risk!**

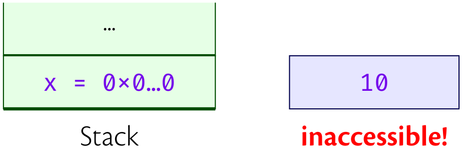
Garbage

- Dangling pointers are a serious problem!
- Accessing a deallocated location could give arbitrary results
- **Huge security risk!**
- Garbage is not so serious, but wastes resources!

Garbage

- Dangling pointers are a serious problem!
- Accessing a deallocated location could give arbitrary results
- **Huge security risk!**
- Garbage is not so serious, but wastes resources!
- Can happen even with explicit deallocation

```
int *x = malloc(
    sizeof(int));
*x = 10;
x = NULL;
```



Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!

Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!
- Takes explicit deallocation out of the programmer's hands

Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!
- Takes explicit deallocation out of the programmer's hands
- The problem of dangling references is avoided

Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!
- Takes explicit deallocation out of the programmer's hands
- The problem of dangling references is avoided
- The run-time support takes the responsibility of identifying and deallocating garbage

Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!
- Takes explicit deallocation out of the programmer's hands
- The problem of dangling references is avoided
- The run-time support takes the responsibility of identifying and deallocating garbage
- Garbage is any block of memory not directly or indirectly accessible from the stack

Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!
- Takes explicit deallocation out of the programmer's hands
- The problem of dangling references is avoided
- The run-time support takes the responsibility of identifying and deallocating garbage
- Garbage is any block of memory not directly or indirectly accessible from the stack
- The system maintains a **free list** – blocks of memory that available to be allocated

Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!
- Takes explicit deallocation out of the programmer's hands
- The problem of dangling references is avoided
- The run-time support takes the responsibility of identifying and deallocating garbage
- Garbage is any block of memory not directly or indirectly accessible from the stack
- The system maintains a **free list** – blocks of memory that available to be allocated
- Garbage collection – identify garbage and add to the free list

Automatic garbage collection

- Java and modern languages like Swift provide **automatic garbage collection**!
- Takes explicit deallocation out of the programmer's hands
- The problem of dangling references is avoided
- The run-time support takes the responsibility of identifying and deallocating garbage
- Garbage is any block of memory not directly or indirectly accessible from the stack
- The system maintains a **free list** – blocks of memory that available to be allocated
- Garbage collection – identify garbage and add to the free list
- Two broad methods – **mark-and-sweep** and **reference counting**

Mark-and-sweep

Mark phase

```
//  all nodes are white
shade roots;
while (there are gray nodes) {
    choose a gray node n;
    shade n->left and n->right;
    n.color = black;
}
//  no gray nodes
//  reachable nodes black
shade(n): if n.color == white
        n.color = gray;
```

- The mark phase blackens all nodes reachable from the roots (nodes on the stack)

Mark-and-sweep

Mark phase

```
//  all nodes are white
shade roots;
while (there are gray nodes) {
    choose a gray node n;
    shade n→left and n→right;
    n.color = black;
}
//  no gray nodes
//  reachable nodes black
shade(n): if n.color == white
        n.color = gray;
```

- The mark phase blackens all nodes reachable from the roots (nodes on the stack)
- The unreachable nodes are white

Mark-and-sweep

Mark phase

```
//  all nodes are white
shade roots;
while (there are gray nodes) {
    choose a gray node n;
    shade n→left and n→right;
    n.color = black;
}
//  no gray nodes
//  reachable nodes black
shade(n): if n.color == white
        n.color = gray;
```

- The mark phase blackens all nodes reachable from the roots (nodes on the stack)
- The unreachable nodes are white
- The **sweep phase** goes through all memory locations in order

Mark-and-sweep

Mark phase

```
//  all nodes are white
shade roots;
while (there are gray nodes) {
    choose a gray node n;
    shade n→left and n→right;
    n.color = black;
}
//  no gray nodes
//  reachable nodes black
shade(n): if n.color == white
        n.color = gray;
```

- The mark phase blackens all nodes reachable from the roots (nodes on the stack)
- The unreachable nodes are white
- The **sweep phase** goes through all memory locations in order
- If the node is black, color it white and add it to free list

Mark-and-sweep

Mark phase

```
//  all nodes are white
shade roots;
while (there are gray nodes) {
    choose a gray node n;
    shade n→left and n→right;
    n.color = black;
}
//  no gray nodes
//  reachable nodes black
shade(n): if n.color == white
        n.color = gray;
```

- The mark phase blackens all nodes reachable from the roots (nodes on the stack)
- The unreachable nodes are white
- The **sweep phase** goes through all memory locations in order
- If the node is black, color it white and add it to free list
- If the node is white, leave it as it is

Mark-and-sweep

Mark phase

```
//  all nodes are white
shade roots;
while (there are gray nodes) {
    choose a gray node n;
    shade n→left and n→right;
    n.color = black;
}
//  no gray nodes
//  reachable nodes black
shade(n): if n.color == white
        n.color = gray;
```

- The mark phase blackens all nodes reachable from the roots (nodes on the stack)
- The unreachable nodes are white
- The **sweep phase** goes through all memory locations in order
- If the node is black, color it white and add it to free list
- If the node is white, leave it as it is
- **Garbage collection is run while pausing other programs!**

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!
- Freshly allocated objects are located in a **nursery**

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!
- Freshly allocated objects are located in a **nursery**
- GC is run frequently on the nursery

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!
- Freshly allocated objects are located in a **nursery**
- GC is run frequently on the nursery
- Each time an object in the nursery survives a GC run, its generation is incremented

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!
- Freshly allocated objects are located in a **nursery**
- GC is run frequently on the nursery
- Each time an object in the nursery survives a GC run, its generation is incremented
- If it survives some number of generations, it is moved to the main heap

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!
- Freshly allocated objects are located in a **nursery**
- GC is run frequently on the nursery
- Each time an object in the nursery survives a GC run, its generation is incremented
- If it survives some number of generations, it is moved to the main heap
- GC on the main heap is run much less frequently as turnover is much less!

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!
- Freshly allocated objects are located in a **nursery**
- GC is run frequently on the nursery
- Each time an object in the nursery survives a GC run, its generation is incremented
- If it survives some number of generations, it is moved to the main heap
- GC on the main heap is run much less frequently as turnover is much less!
- Such **generational garbage collectors** have much higher throughput!

Improving garbage collection

- Improvements to mark-and-sweep GC using the **generational hypothesis**
  - Most objects die young (as soon as the current stack frame is deallocated)
  - But if something survives for a certain time it is likely to live long!
- Freshly allocated objects are located in a **nursery**
- GC is run frequently on the nursery
- Each time an object in the nursery survives a GC run, its generation is incremented
- If it survives some number of generations, it is moved to the main heap
- GC on the main heap is run much less frequently as turnover is much less!
- Such **generational garbage collectors** have much higher throughput!
- Each GC run will return a higher fraction of the visited nodes to the free list

Reference counting

- Languages like **Swift** use automatic reference counting

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block
- If an assignment introduces a new reference to a block, increment the counter

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block
- If an assignment introduces a new reference to a block, increment the counter
- If an assignment makes a reference point away from a location, decrement the counter

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block
- If an assignment introduces a new reference to a block, increment the counter
- If an assignment makes a reference point away from a location, decrement the counter
- If the counter of node n hits zero …

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block
- If an assignment introduces a new reference to a block, increment the counter
- If an assignment makes a reference point away from a location, decrement the counter
- If the counter of node n hits zero …
- decrement the counter of all nodes pointed to by n and return n to free list!

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block
- If an assignment introduces a new reference to a block, increment the counter
- If an assignment makes a reference point away from a location, decrement the counter
- If the counter of node n hits zero …
- decrement the counter of all nodes pointed to by n and return n to free list!
- There are no long pauses due to running garbage collection

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block
- If an assignment introduces a new reference to a block, increment the counter
- If an assignment makes a reference point away from a location, decrement the counter
- If the counter of node n hits zero …
- decrement the counter of all nodes pointed to by n and return n to free list!
- There are no long pauses due to running garbage collection
- Maintaing the reference count is mostly low-cost!

Reference counting

- Languages like **Swift** use automatic reference counting
- Every memory block has some bits reserved to maintain a counter
- Keeps track of the number of references into the memory block
- If an assignment introduces a new reference to a block, increment the counter
- If an assignment makes a reference point away from a location, decrement the counter
- If the counter of node n hits zero …
- decrement the counter of all nodes pointed to by n and return n to free list!
- There are no long pauses due to running garbage collection
- Maintaing the reference count is mostly low-cost!
- Infrequently, **a decrement to the reference count might trigger a chain of deallocations**

Reference counting and cycles

- Reference counting does not handle cycles properly

Reference counting and cycles

- Reference counting does not handle cycles properly
- Consider a list of nodes $n_0, \ldots, n_k$, with each n_i pointing to n_{i+1} and n_k points to n_0

Reference counting and cycles

- Reference counting does not handle cycles properly
- Consider a list of nodes $n_0, \ldots, n_k$, with each n_i pointing to n_{i+1} and n_k points to n_0
- Suppose none of them are reachable from the stack

Reference counting and cycles

- Reference counting does not handle cycles properly
- Consider a list of nodes $n_0, \ldots, n_k$, with each n_i pointing to n_{i+1} and n_k points to n_0
- Suppose none of them are reachable from the stack
- The counters of all these nodes never reach 0

Reference counting and cycles

- Reference counting does not handle cycles properly
- Consider a list of nodes $n_0, \ldots, n_k$, with each n_i pointing to n_{i+1} and n_k points to n_0
- Suppose none of them are reachable from the stack
- The counters of all these nodes never reach 0
- **Cyclic garbage is not reclaimed!**

Reference counting and cycles

- Reference counting does not handle cycles properly
- Consider a list of nodes $n_0, \ldots, n_k$, with each n_i pointing to n_{i+1} and n_k points to n_0
- Suppose none of them are reachable from the stack
- The counters of all these nodes never reach 0
- **Cyclic garbage is not reclaimed!**
- Can be handled by running mark-and-sweep at less frequent intervals

Swift and weak references

- **Swift** reduces cyclic garbage by supporting **weak references** in programs

Swift and weak references

- **Swift** reduces cyclic garbage by supporting **weak references** in programs
- An example usage …

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
    }
    var apartment: Apartment?
}
```

```
class Apartment {
    let unit: String
    init(unit: String) {
        self.unit = unit
    }
    weak var tenant: Person?
}
```

Swift's ARC (automatic reference counting)

- Swift encourages programmers to prevent obvious reference cycles by making some references weak (typically from a "subordinate object" to a superior object)

Swift's ARC (automatic reference counting)

- Swift encourages programmers to prevent obvious reference cycles by making some references weak (typically from a "subordinate object" to a superior object)
- Weak references are always variables with an **optional** type (like **`Maybe`** in Haskell)

Swift's ARC (automatic reference counting)

- Swift encourages programmers to prevent obvious reference cycles by making some references weak (typically from a "subordinate object" to a superior object)
- Weak references are always variables with an **optional** type (like **`Maybe`** in Haskell)
- Optional values can be **`nil`**, so we cannot use them without checking if it is non-nil

Swift's ARC (automatic reference counting)

- Swift encourages programmers to prevent obvious reference cycles by making some references weak (typically from a "subordinate object" to a superior object)
- Weak references are always variables with an **`optional`** type (like **`Maybe`** in Haskell)
- Optional values can be **`nil`**, so we cannot use them without checking if it is non-nil
- Weak references do not contribute the reference count of a node – only strong references are counted

Swift's ARC (automatic reference counting)

- Swift encourages programmers to prevent obvious reference cycles by making some references weak (typically from a "subordinate object" to a superior object)
- Weak references are always variables with an **optional** type (like `Maybe` in Haskell)
- Optional values can be `nil`, so we cannot use them without checking if it is non-nil
- Weak references do not contribute the reference count of a node – only strong references are counted
- So a node can be deallocated if there is no strong reference pointing to it

Swift's ARC (automatic reference counting)

- Swift encourages programmers to prevent obvious reference cycles by making some references weak (typically from a "subordinate object" to a superior object)
- Weak references are always variables with an **optional** type (like **`Maybe`** in Haskell)
- Optional values can be **`nil`**, so we cannot use them without checking if it is non-nil
- Weak references do not contribute the reference count of a node – only strong references are counted
- So a node can be deallocated if there is no strong reference pointing to it
- Swift's **ARC** automatically changes weak references to deallocated nodes to **`nil`**

Swift's ARC (automatic reference counting)

- Swift encourages programmers to prevent obvious reference cycles by making some references weak (typically from a "subordinate object" to a superior object)
- Weak references are always variables with an **optional** type (like **`Maybe`** in Haskell)
- Optional values can be **`nil`**, so we cannot use them without checking if it is non-nil
- Weak references do not contribute the reference count of a node – only strong references are counted
- So a node can be deallocated if there is no strong reference pointing to it
- Swift's **ARC** automatically changes weak references to deallocated nodes to **`nil`**
- Because programmers cannot use optional values without checking if they are non-nil, there is no safety issues due to dangling pointers