Memory management

Or: why C++ is bad for your brain

Koz Ross

18th May, 2017

[AUT CSC]

AUT Computer Science Club

Outline

Introduction

Preliminaries

The three management approaches

Advantages and disadvantages

Questions

Why memory management?

- All programming ultimately comes back to memory

Why memory management?

- All programming ultimately comes back to memory
- Thanks to the memory wall getting ever higher, how we work with memory is more important than ever

Why memory management?

- All programming ultimately comes back to memory
- Thanks to the memory wall getting ever higher, how we work with memory is more important than ever
- New programming languages appear all the time — knowing how memory is managed will give you an edge in deciding if it's any good

Why memory management?

- All programming ultimately comes back to memory
- Thanks to the memory wall getting ever higher, how we work with memory is more important than ever
- New programming languages appear all the time — knowing how memory is managed will give you an edge in deciding if it's any good
- If you ever have to *implement* a language, this stuff is worth knowing!

Some terminology

- When we talk about *memory* here, we refer to memory which is:

Some terminology

- When we talk about *memory* here, we refer to memory which is:
    - *Volatile* (i.e. lasts only as long as the program)

Some terminology

- When we talk about *memory* here, we refer to memory which is:
  - *Volatile* (i.e. lasts only as long as the program)
  - *Writeable* (i.e. we can change it, possibly several times)

Some terminology

- When we talk about *memory* here, we refer to memory which is:
    - *Volatile* (i.e. lasts only as long as the program)
    - *Writeable* (i.e. we can change it, possibly several times)
- When we *allocate* memory, we reserve it for our use

Some terminology

- When we talk about *memory* here, we refer to memory which is:
    - *Volatile* (i.e. lasts only as long as the program)
    - *Writeable* (i.e. we can change it, possibly several times)
- When we *allocate* memory, we reserve it for our use
- When we don't need it anymore, we *deallocate* it, releasing it for someone else to play with (possibly even ourselves, for later)

Some terminology

- When we talk about *memory* here, we refer to memory which is:
  - *Volatile* (i.e. lasts only as long as the program)
  - *Writeable* (i.e. we can change it, possibly several times)
- When we *allocate* memory, we reserve it for our use
- When we don't need it anymore, we *deallocate* it, releasing it for someone else to play with (possibly even ourselves, for later)
- A *reference* is a marker for a location in memory (basically, a C pointer)

Two kinds of memory

Two kinds of memory

- *Scoped:*

Two kinds of memory

- *Scoped:*
    - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately

Two kinds of memory

- *Scoped:*
  - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately
  - Smaller (relatively); used for single variables or small data structures which aren't needed for long

Two kinds of memory

- *Scoped:*
  - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately
  - Smaller (relatively); used for single variables or small data structures which aren't needed for long
  - Sometimes called 'automatic', or 'stack', memory (but we won't use those terms)

Two kinds of memory

- *Scoped:*
  - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately
  - Smaller (relatively); used for single variables or small data structures which aren't needed for long
  - Sometimes called 'automatic', or 'stack', memory (but we won't use those terms)
- *Non-scoped:*

Two kinds of memory

- *Scoped:*
  - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately
  - Smaller (relatively); used for single variables or small data structures which aren't needed for long
  - Sometimes called 'automatic', or 'stack', memory (but we won't use those terms)
- *Non-scoped:*
  - Can persist past the scope in which it was allocated

Two kinds of memory

- *Scoped:*
  - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately
  - Smaller (relatively); used for single variables or small data structures which aren't needed for long
  - Sometimes called 'automatic', or 'stack', memory (but we won't use those terms)
- *Non-scoped:*
  - Can persist past the scope in which it was allocated
  - Larger (again, relatively); used for bigger structures that need to be around for a long time

Two kinds of memory

- *Scoped:*
  - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately
  - Smaller (relatively); used for single variables or small data structures which aren't needed for long
  - Sometimes called 'automatic', or 'stack', memory (but we won't use those terms)
- *Non-scoped:*
  - Can persist past the scope in which it was allocated
  - Larger (again, relatively); used for bigger structures that need to be around for a long time
  - Sometimes called 'manual', or 'heap', memory (but we won't use *those* terms either)

Two kinds of memory

- *Scoped:*
  - Tied to the scope in which it was allocated; once we leave that scope, the memory gets deallocated immediately
  - Smaller (relatively); used for single variables or small data structures which aren't needed for long
  - Sometimes called 'automatic', or 'stack', memory (but we won't use those terms)
- *Non-scoped:*
  - Can persist past the scope in which it was allocated
  - Larger (again, relatively); used for bigger structures that need to be around for a long time
  - Sometimes called 'manual', or 'heap', memory (but we won't use *those* terms either)

Out of the two, scoped memory is usually quite simple. For *non-scoped* memory, there are a few added challenges and trade-offs...

Challenges with non-scoped memory

- How much of the responsibility for deallocating the memory is on the programmer?

Challenges with non-scoped memory

- How much of the responsibility for deallocating the memory is on the programmer?
- How predictable should it be?

Challenges with non-scoped memory

- How much of the responsibility for deallocating the memory is on the programmer?
- How predictable should it be?
- What costs are we willing to pay for it?

Challenges with non-scoped memory

- How much of the responsibility for deallocating the memory is on the programmer?
- How predictable should it be?
- What costs are we willing to pay for it?

Programming languages have converged on three different approaches to non-scoped memory management.

Manual management

Manual management

- The programmer must explicitly deallocate all non-scoped memory themselves (and deal with all the attendant issues)

Manual management

- The programmer must explicitly deallocate all non-scoped memory themselves (and deal with all the attendant issues)
- Made famous by the C programming language, although C wasn't the first to use that method

Manual management

- The programmer must explicitly deallocate all non-scoped memory themselves (and deal with all the attendant issues)
- Made famous by the C programming language, although C wasn't the first to use that method
- Also a major reason why people hate writing C

Reference counting

Reference counting

- When we allocate a block of non-scoped memory, we also make a counter for the number of references we have to that block (a *refcount*)

Reference counting

- When we allocate a block of non-scoped memory, we also make a counter for the number of references we have to that block (a *refcount*)
- Every time we take a new reference to a block of non-scoped memory, we increment that block's refcount; when that reference goes away, we decrement the correspoding refcount

Reference counting

- When we allocate a block of non-scoped memory, we also make a counter for the number of references we have to that block (a *refcount*)
- Every time we take a new reference to a block of non-scoped memory, we increment that block's refcount; when that reference goes away, we decrement the correspoding refcount
- When a block's refcount reaches 0, we know that we can't reach it anymore, and the block gets deallocated

Garbage collection

Garbage collection

- Every reference, and all memory, is tracked by the programming language runtime automatically

Garbage collection

- Every reference, and all memory, is tracked by the programming language runtime automatically
- When the runtime determines that a block of non-scoped memory has no references left to it, it will mark it as unused

Garbage collection

- Every reference, and all memory, is tracked by the programming language runtime automatically
- When the runtime determines that a block of non-scoped memory has no references left to it, it will mark it as unused
- At some later point, the runtime will deallocate all the unused blocks that exist at the time

Answering the challenges

	Responsibility	**Predictability**	**Relative cost**
Manual	All programmer	Total	Low
Refcounting	Some programmer	Moderate	Moderate
GC	All language	Low	High

Answering the challenges

	Responsibility	**Predictability**	**Relative cost**
Manual	All programmer	Total	Low
Refcounting	Some programmer	Moderate	Moderate
GC	All language	Low	High

This is a bit non-specific — let's consider each one separately.

Manual management tradeoffs

- Benefits:

Manual management tradeoffs

- Benefits:
  - Lowest resource cost (basically none)

Manual management tradeoffs

- Benefits:
    - Lowest resource cost (basically none)
    - Simplest for the language implementation and runtime

Manual management tradeoffs

- Benefits:
  - Lowest resource cost (basically none)
  - Simplest for the language implementation and runtime
  - Completely predictable (thus, can be very heavily hand-optimized)

Manual management tradeoffs

- Benefits:
  - Lowest resource cost (basically none)
  - Simplest for the language implementation and runtime
  - Completely predictable (thus, can be very heavily hand-optimized)
- Drawbacks:

Manual management tradeoffs

- Benefits:
  - Lowest resource cost (basically none)
  - Simplest for the language implementation and runtime
  - Completely predictable (thus, can be very heavily hand-optimized)
- Drawbacks:
  - *Painfully* tedious and error-prone (memory leaks, dangling pointers, wild pointers,...)

Manual management tradeoffs

- Benefits:
  - Lowest resource cost (basically none)
  - Simplest for the language implementation and runtime
  - Completely predictable (thus, can be very heavily hand-optimized)
- Drawbacks:
  - *Painfully* tedious and error-prone (memory leaks, dangling pointers, wild pointers,...)
  - Vulnerable to external memory fragmentation

Manual management tradeoffs

- Benefits:
  - Lowest resource cost (basically none)
  - Simplest for the language implementation and runtime
  - Completely predictable (thus, can be very heavily hand-optimized)
- Drawbacks:
  - *Painfully* tedious and error-prone (memory leaks, dangling pointers, wild pointers,...)
  - Vulnerable to external memory fragmentation
  - *Requires* manual tuning to be effective (whether you know how or not)

Unsurprisingly, manual management is unpopular — no programming language invented since the *eighties* uses it by default.

Manual management tradeoffs

- Benefits:
  - Lowest resource cost (basically none)
  - Simplest for the language implementation and runtime
  - Completely predictable (thus, can be very heavily hand-optimized)
- Drawbacks:
  - *Painfully* tedious and error-prone (memory leaks, dangling pointers, wild pointers,...)
  - Vulnerable to external memory fragmentation
  - *Requires* manual tuning to be effective (whether you know how or not)

Unsurprisingly, manual management is unpopular — no programming language invented since the *eighties* uses it by default. It was used more frequently before, as we had fewer alternatives and *much* more constrained computing resources (the computer C was designed on had 16K of RAM!).

Reference counting tradeoffs

- Benefits:

Reference counting tradeoffs

- Benefits:
  - Much less tedious

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead
  - Fairly simple for the language implementation and runtime

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead
  - Fairly simple for the language implementation and runtime
- Drawbacks:

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead
  - Fairly simple for the language implementation and runtime
- Drawbacks:
  - Vulnerable to cyclic references

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead
  - Fairly simple for the language implementation and runtime
- Drawbacks:
  - Vulnerable to cyclic references
  - Vulnerable to external memory fragmentation

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead
  - Fairly simple for the language implementation and runtime
- Drawbacks:
  - Vulnerable to cyclic references
  - Vulnerable to external memory fragmentation
  - Very bad in heavily-concurrent environments

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead
  - Fairly simple for the language implementation and runtime
- Drawbacks:
  - Vulnerable to cyclic references
  - Vulnerable to external memory fragmentation
  - Very bad in heavily-concurrent environments

Refcounting was first made famous by C++: initially, it had to be done by hand, but now has language-level support.

Reference counting tradeoffs

- Benefits:
  - Much less tedious
  - Deterministic and fairly predictable
  - Imposes very little overhead
  - Fairly simple for the language implementation and runtime
- Drawbacks:
  - Vulnerable to cyclic references
  - Vulnerable to external memory fragmentation
  - Very bad in heavily-concurrent environments

Refcounting was first made famous by C++: initially, it had to be done by hand, but now has language-level support. Surprisingly unpopular — mostly used by pure functional languages like Haskell or C++ derivatives like Swift and Rust.

Garbage collection tradeoffs

- Benefits:

Garbage collection tradeoffs

- Benefits:
    - Very easy on the programmer (basically *never* have to think about memory management)

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention
- Drawbacks:

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention
- Drawbacks:
  - Trickiest for the runtime (especially for more modern garbage collectors)

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention
- Drawbacks:
  - Trickiest for the runtime (especially for more modern garbage collectors)
  - Very unpredictable in general

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention
- Drawbacks:
  - Trickiest for the runtime (especially for more modern garbage collectors)
  - Very unpredictable in general
  - (Relatively) resource-intensive

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention
- Drawbacks:
  - Trickiest for the runtime (especially for more modern garbage collectors)
  - Very unpredictable in general
  - (Relatively) resource-intensive

GC has maintained a rabid cult of detractors since it was invented in the 1950s (most of whom write C++).

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention
- Drawbacks:
  - Trickiest for the runtime (especially for more modern garbage collectors)
  - Very unpredictable in general
  - (Relatively) resource-intensive

GC has maintained a rabid cult of detractors since it was invented in the 1950s (most of whom write C++). Many of their critiques are hilariously outdated or irrelevant today.

Garbage collection tradeoffs

- Benefits:
  - Very easy on the programmer (basically *never* have to think about memory management)
  - Can deal with external memory fragmentation and high contention without programmer intervention
- Drawbacks:
  - Trickiest for the runtime (especially for more modern garbage collectors)
  - Very unpredictable in general
  - (Relatively) resource-intensive

GC has maintained a rabid cult of detractors since it was invented in the 1950s (most of whom write C++). Many of their critiques are hilariously outdated or irrelevant today. However, there are still cases where the cost of GC is too high, or where its unpredictability is an issue.

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
    - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
  - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory
  - Out of all the languages invented in the last 10 years, all but two (Swift and Rust) use GC by default

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
  - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory
  - Out of all the languages invented in the last 10 years, all but two (Swift and Rust) use GC by default
  - Even Bjarne Stroustrup (inventor of C++) admitted C++ should have garbage collection (optionally)...

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
  - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory
  - Out of all the languages invented in the last 10 years, all but two (Swift and Rust) use GC by default
  - Even Bjarne Stroustrup (inventor of C++) admitted C++ should have garbage collection (optionally)... nearly *twenty years ago* (take that, C++ cultists!)

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
  - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory
  - Out of all the languages invented in the last 10 years, all but two (Swift and Rust) use GC by default
  - Even Bjarne Stroustrup (inventor of C++) admitted C++ should have garbage collection (optionally)... nearly *twenty years ago* (take that, C++ cultists!)
- However, hype-following is *never* a good idea; a better plan is to consider your specific case and problem and go from there

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
  - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory
  - Out of all the languages invented in the last 10 years, all but two (Swift and Rust) use GC by default
  - Even Bjarne Stroustrup (inventor of C++) admitted C++ should have garbage collection (optionally)... nearly *twenty years ago* (take that, C++ cultists!)
- However, hype-following is *never* a good idea; a better plan is to consider your specific case and problem and go from there
- The costs to GC are getting lower every day, partly because of better hardware, and partly because of better collectors

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
  - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory
  - Out of all the languages invented in the last 10 years, all but two (Swift and Rust) use GC by default
  - Even Bjarne Stroustrup (inventor of C++) admitted C++ should have garbage collection (optionally)... nearly *twenty years ago* (take that, C++ cultists!)
- However, hype-following is *never* a good idea; a better plan is to consider your specific case and problem and go from there
- The costs to GC are getting lower every day, partly because of better hardware, and partly because of better collectors
- In short: *know your tradeoffs!*

So, which should we use?

- Popularity-contest-wise, GC wins by a landslide:
  - Out of the TIOBE index's top 20 languages, *twelve* use GC as their default method for managing non-scoped memory
  - Out of all the languages invented in the last 10 years, all but two (Swift and Rust) use GC by default
  - Even Bjarne Stroustrup (inventor of C++) admitted C++ should have garbage collection (optionally)... nearly *twenty years ago* (take that, C++ cultists!)
- However, hype-following is *never* a good idea; a better plan is to consider your specific case and problem and go from there
- The costs to GC are getting lower every day, partly because of better hardware, and partly because of better collectors
- In short: *know your tradeoffs!* (Well, I guess *don't be a cultist* also suits...)

Questions?