Overview

Dangers of explicit deallocation
...
Introduction to automatic memory management
Reference counting

**BREAK**

Tracing collectors
Automatic memory management in C++
Finalisation
Advanced techniques: generational and incremental GC
What is the problem?

Memory is a scarce and precious resource

Some applications can manage with a bounded amount of memory using static allocation combined with stack allocation.

Others use dynamic allocation of memory because:

- Some objects live longer than the method that creates them.
- Recursive data structures such as lists and trees.
- Avoids fixed hard limits on data structure sizes.

If we had unbounded amounts of memory, we’d never worry.

The *PROBLEM* is that we don’t have *unbounded* memory.
What can we do?

Dynamically allocate and deallocate memory.

REUSE deallocated memory.

Dynamical memory allocation is available in many languages, e.g., using languages features:

- **New** allocates a new object
- **Delete $X$** deallocated the object $X$

Such features allows programmer to handle allocation themselves.

Objects that no longer are needed are called garbage.
Garbage Collection

Identifying garbage and deallocating the memory it occupies is called *garbage collection*.

We can try to handle the garbage collection housekeeping chores related to object allocation and deallocation ourselves.

Such housekeeping can be simple but for many applications the chores become complex – and error prone.

*Can we do it ourselves? – Or should it be automatic??*
Because human programmers just can’t get it right.
Either

too little is collected leading to memory leaks, or

too much is collected leading to broken programs.
Space leaks?

Human programmers can:
- Forget to delete an object when it is no longer needed.
- Return a newly allocated object – but when will it be deallocated?
- Not figure out when a shared objects should be deleted.

Sharing is a significant problem
Can be handled by using the principle *last one to leave the room turns off the light.*
However, this is easily forgotten, and, worse, in a large building, it can be close to impossible to detect that you are the last!
Dangling Pointers

Eager human programmers can delete objects too early leading to *dangling pointers*

Consider an object that is shared between two different parts of a program each having its own pointer to the object.

If one of the pointers is deleted then the other pointer is left pointing to a *non-existent object* – we say that it is a *dangling pointer*. 
Besides the practical problems of explicit memory management, we also believe that explicit management conflicts with the software engineering principles of **abstraction** and **modularity**.

**THEREFORE**
this lecture is about building automatic garbage collectors

**BUT ALSO** about how to survive without automatic GC
More “why”

Don’t be hard on yourself

• Don’t reinvent the wheel
• Garbage collectors honed by time and much usage can offer better performance than custom memory managers

Caveat

• It’s not a silver bullet
• Some memory management problems cannot be solved using automatic GC, e.g. if you forget to drop references to objects that you no longer need.
• Some environments are inimicable to garbage collection
  – embedded systems with limited memory
  – hard real-time systems
Part 2: Object Allocation

In the following, we review how objects are allocated.

- Object & Machine Model?
- Explicit Allocation
- Dangers of Explicit Allocation
Object Model

In this tutorial, we assume that we are using a system that is to support some kind of objects.

Objects are represented in memory by some number of bits stored contiguously in memory. They consist of a header (which is not visible to the user program — often called the mutator) and zero or more fields.

Objects can contain references to other objects.

A reference to an object is usually implemented merely by the memory address of the piece of memory where the object is stored.

For simplicity, we assume that there is only one thread of execution.
Machine Model

Our object system runs on some machine consisting of:

- A *stack* which implements the executing thread of control
- A part of memory where *global variables* are stored
- A number of *registers* that contain addresses of various parts of memory (or arithmetic data)
- A *heap* which is a part of memory that is split into many pieces each of which either contains the representation of an object or is unused (in which case we say it is *free*).
Allocation

Allocation means finding a free piece of memory in the heap and reserving it for the representation of an object.

Deallocation means changing the status of a piece of memory from allocated to free.

Liveness An object is live as long as it still is reachable from some part of the program’s computation.
Objects in Memory

Objects are typically allocated in the heap.

An Object Reference is a pointer to an object (typically merely the heap address of the start of the object).

Variables contain object references (ignoring primitive data).

Each object can contain a number of variables and thereby reference other objects.
Static & dynamic allocation

*Static allocation* — allocation takes place when a program starts – basically memory is laid out by the compiler.

*Dynamic allocation* — allocate new objects while the program is executing.

A simple form of dynamic allocation is *stack allocation* where objects are allocated on the program stack and deallocated using a stack discipline.

Heap allocation is the most general form of allocation: objects are allocated in the heap.
Dangers of Explicit Deallocation

With explicit deallocation the programmer ends up:

Doing too little

- Garbage objects are not deallocated and slowly but surely clutters memory and so the program runs out of memory (such a failure to delete garbage objects is called a memory leak).

Doing too much

- Throwing away a non-garbage object. Subsequent use of a live reference to the object will cause the program to fail in inexplicable ways. Such a reference is a dangling reference.

- Throwing away a garbage object twice! Likely to break the memory manager.
The **REAL** bad thing about explicit deallocation

The problems of:

- Memory leaks
- Dangling references
- Double deallocation

are real and omnipresent in explicit deallocation systems and they cause the real problem:

**Wasting huge amounts of debugging time!**

and despite this, programs may still fail in mysterious ways long after being put into production.

Finding and fixing MM bugs can account for 40% of debug time.
Part 3: Living without GC: Doing it yourself

How do we manage without GC?
In the following, we take a look at techniques for doing it yourself or with some help from tools.

- Defensive programming
- Pairing Principle
- The Ownership concept
- Monitoring technique
- Administrator technique
- Tools
- Living without GC in C++
Defensive programming

Defensive strategies for doing without GC

Sharing:

- Copying objects instead of sharing.
- Transform a global deallocation decision to a local one

Can be wasteful of space, but can be useful.

Example:
Everyone gets their own set of lights: turn your lights off when leaving.  (Simple, but obviously wasteful.)
Pairing Principle

Pairing principle:
For each `new()` pair it with a `delete()`
Make sure there is a one-to-one correspondence.
For every `new()` check that the corresponding `delete()` is there.
One way is to have the `allocating` object also be the `deallocate` object.

Example

_If you turned the light on, YOU turn it off._
Pairing Principle Example

Allocate in the constructor; deallocate in the destructor.

Class A {
    Xclass X;
    void A() { X = new Xclass;}
    void ~A() { delete X;}
    ...
}

Ownership Concept

Observation: Objects are often passed around
Thus it is often an object other than the creator who must do the deallocation.

Ownership Concept
Initially, the allocating object is the owner of the newly allocated object. When passing a reference to the allocated object, the ownership can also be passed. (The previous owner should throw away the reference – it may become dangling very soon!)

Only the owner is allowed to deallocate the object; the last owner does the deallocation.
Each owner either passes on the ownership rights – or deallocates.
Monitoring Technique

Monitoring Tool
A simple mechanism to help find bugs is to maintain a table of allocated objects.

`Malloc()` is replaced by a version that store the address of newly allocated objects in a table. `Free()` is replaced by a version that checks the table before freeing.

Such monitoring can help find bugs:
- memory leaks (the table will fill with a particular type of object),
- dangling references (the table can be checked to see, if a reference is valid before using it)
- double deallocations (free will protest if a non-allocated object is free’d).
Multiple Owners: Shared Objects

Handling Shared Objects Using Reference Counting

We could attempt to handle shared objects by trying to keep track of multiple owners, e.g., by expanding the monitoring table by a count field.

For every new owner, we must increment the count. And decrement it every time an owner is done with the object.

When the last owner is done (the count goes to zero), the object is deallocated.

Requires extra code for deallocation, allocation, copying of references, etc.
Part 4: Automatic Memory Management

Automatic memory management including garbage collection handles the most significant of the problems that we tried to solve until now. Doing it yourself has is cumbersome to do – and quite error prone. In the following, we present automatic memory management
What is garbage?

Almost all garbage collectors assume the following definition of live objects called liveness by reachability: if you can get to an object, then it is live.

More formally: An object is live if and only if:
- it is referenced in a predefined variable called a root,
- or
- it is referenced in a variable contained in a live object (i.e. it is transitively referenced from a root).

Non-live objects are called dead objects, i.e. garbage.
The objects and references can be considered a directed graph: The live objects of the graph are those reachable from a root. The process executing a computation is called the mutator because it is viewed as dynamically changing the object graph.

What are the roots of a computation?

Determining roots is, in general, language-dependent

In common language implementations roots include

- words in the static area
- registers
- words on the execution stack that point into the heap.
Why garbage collect?

Language requirement

• many OO languages assume GC, e.g. allocated objects may survive much longer than the method that created them

Problem requirement

• the nature of the problem may make it very hard/impossible to determine when something is garbage
Why GC is a software engineering issue

SE = management of complexity in large-scale software systems

Tools: modularity & abstraction
- Explicit MM cuts against these principles
- Auto MM offers increased abstraction

Relieve programmers of book-keeping detail
- Time is better spent on higher-level details of design and implementation
Reliable code is understandable code

Understand behaviour of a module or a few neighbouring modules

Behaviour of module should be independent of context

One module should not cause the failure of another (e.g. through a MM error)
Composing components

Modules should be reusable in different contexts

- Cohesive
- Loosely-coupled
- Communicate with as few other modules as possible and exchange as little information as possible [Meyer]
- Interfaces should be simple and well-defined
Liveness is a global property

- An object is live if it can be used by *any* part of the program
- This cannot (in general) be determined by inspection of a single code fragment

Adding MM book-keeping clutter to interfaces

- Weakens abstractions
- Reduces extensibility of modules
When stack A is popped, can first data be reclaimed?
**Liveness**

An object is only live if it can effect future computation

- Must be able to load it (or a part of it) into registers.
- Well-behaved programs that do not access random addresses in memory.

What data is known (and can be manipulated)?

- Global data held in static areas
- Local variables, parameters and compiler temporaries that may be held on the stack or in machine registers

Hence the program may also use

- Any objects that can be reached by way computations on known objects.
Liveness by reachability

Almost all garbage collectors assume the following definition of live objects called **liveness by reachability**: if you can get to an object, then it is live.

More formally: An object is **live** if and only if:

- it is referenced in a predefined variable called a **root**, or
- it is referenced in a variable contained in a live object (i.e. it is transitively referenced from a root).

Non-live objects are called **dead** objects, i.e. **garbage**.
A conservative estimate

‘Liveness by reachability’ provides a conservative estimate of the set of live objects.

- Contains all objects that could be used by a well-behaved program
- May contain objects that will never be used again.

```java
Thing a = someComputation();
if(a.property())
  E1();
else
  E2();
```

Reference to `a` may be held on stack — hence considered reachable — until `E1/E2` has completed.

But static analysis may reveal that `a` could be discarded after the conditional test.
Help the GC that you are finished with an object

```java
r = new FileReader(filename)
// use the reader
...
reader.close();
reader = null;
```

- This is a simple, *local*, decision
- *Don’t null* the reference if it is about to disappear (e.g. local variable in a method that’s about to return),
- *Do dispose* of components when you have finished with them.
Cost Metrics for GC

**Execution time**
- total execution time
- distribution of GC execution time
- time to allocate a new object

**Memory usage**
- additional memory overhead
- fragmentation
- virtual memory and cache performance

**Delay time**
- length of disruptive pauses
- zombie times

**Other important metrics**
- comprehensivenesseness
- implementation *simplicity* and *robustness*
No silver bullet

Often not necessary for simple programs
  • But beware reuse of simple code

Hard real-time code

GC doesn’t cure problem of data structures that grow without limit
  • Surprisingly common e.g. caching
  • Benign in small problems, bad for large or long running ones
The basic algorithms

- **Reference counting**: Keep a note on each object in your garage, indicating the number of live references to the object. If an object’s reference count goes to zero, throw the object out (it’s dead).

- **Mark-Sweep**: Put a note on objects you need (roots). Then recursively put a note on anything needed by a live object. Afterwards, check all objects and throw out objects without notes.

- **Mark-Compact**: Put notes on objects you need (as above). Move anything with a note on it to the back of the garage. Burn everything at the front of the garage (it’s all dead).

- **Copying**: Move objects you need to a new garage. Then recursively move anything needed by an object in the new garage. Afterwards, burn down the old garage (any objects in it are dead)!
Reference counting

A mechanism to share ownership

Goal

- identify when you are the *only* owner
- You can make the disposal decision.

Basic idea: count the number of references from live objects.
Reference counting: principle

Each object has a reference count (RC)

- when a reference is copied, the referent’s RC is incremented
- when a reference is deleted, the referent’s RC is decremented
- an object can be reclaimed when its RC = 0

```
Update(left(R), S)
```
Reference counting: recursive freeing

Once an object’s RC=0, it can be freed.

But object may contain references to further objects.

Before this object is freed, the RCs of its constituents should also be freed.
Reference counting: implementation

New() {  
    if free_list == nil  
        abort "Memory exhausted"  
    newcell = allocate()  
    RC(newcell) = 1  
    return newcell  
}

Update(R, S) {  
    RC(S) = RC(S) + 1  
    delete(*R)  
    *R = S  
}

delete(T) {  
    RC(T) = RC(T) - 1  
    if RC(T) == 0 {  
        for U in Children(T)  
            delete(*U)  
            free(T)  
    }  
}

free(N) {  
    next(N) = free_list;  
    free_list = N;  
}
Example

before

after

free
Advantages of reference counting

- Simple to implement
- Costs distributed throughout program
- Good **locality of reference**: only touch old and new targets' RCs
- Works well because few objects are shared and many are short-lived
- **Zombie time** minimized: the zombie time is the time from when an object becomes garbage until it is collected
- Immediate **finalisation** is possible (due to near zero zombie time)
Disadvantages of reference counting

- Not **comprehensive** (does not collect all garbage): *cannot reclaim cyclic data structures*
- High cost of manipulating RCs:
  cost is ever-present even if *no* garbage is collected
- Bad for concurrency; RC manipulations must be atomic — need *Compare&Swap* operation
- Tightly coupled interface to mutator
- High space overheads
- **Recursive freeing** cascade is only bounded by heap size
Tracing GC: idea

We can formalise our definition of reachability:

\[
\text{live} = \{ \, N \in \text{Objects} \mid (\exists \, r \in \text{Roots}. \, r \rightarrow N) \lor (\exists \, M \in \text{live}. \, M \rightarrow N) \, \}\]

We can encode this definition simply

- Start at the roots; the live set is empty
- Add any object that a root points at to our live set
- Repeat
  - Add any object a live object points at to the live set
- Until no new live objects are found
- Any objects not in the live set are garbage
Mark-Sweep

Mark-sweep is such a tracing algorithm — it works by following (tracing) references from live objects to find other live objects.

Implementation of the live set:
Each object has a mark-bit associated with it, indicating whether it is a member of the live set.

There are two phases:

- **Mark phase**: starting from the roots, the graph is traced and the mark-bit is set in each unmarked object encountered. At the end of the mark phase, unmarked objects are garbage.

- **Sweep phase**: starting from the bottom, the heap is swept
  - mark-bit not set: the object is reclaimed
  - mark-bit set: the mark-bit is cleared
The mark-stack

The simplest solution is to implement marking recursively:

- walk a minimum spanning tree of the object graph

```c
mark(N) {
    if markBit(N) == UNMARKED {
        markBit(N) = MARKED
        for M in Children(N)
            mark(*M)
    }
}
```

A more efficient method is to use a marking stack

- Repeat until the mark stack is empty.
  Pop the top item
  If it is unmarked, mark it.
  If it is a branch point in the graph, push any unmarked children onto the stack
New() {  
  if free_pool.empty  
    markHeap()  
    sweep()  
  newobj = allocate()  
  return newobj  
}

markHeap() {  
  markStack = empty  
  for R in Roots  
    markBit(R) = MARKED  
  markStack.push(R)  
  mark()  
}

mark() {  
  while markStack not empty {  
    N = pop(markStack)  
    for M in Children(N)  
      if markBit(M) == UNMARKED {  
        markBit(M) = MARKED  
        if not atom(*M)  
          push(markStack,*M)  
      }  
  }  
}

sweep() {  
  N = Heap_bottom  
  while N < Heap_top  
    if markBit(N) == UNMARKED  
      free (N)  
    else markBit(N) = UNMARKED  
    N += N.size  
}
Marking exercise

- root
- A
- B
- C
- D
- E
- F
- G
- H
- I

mark-bit
Advantages of mark-sweep

✓ Comprehensive: cyclic garbage collected naturally

✓ No run-time overhead on pointer manipulations

✓ Loosely coupled to mutator

✓ Does not move objects
  • does not break any mutator invariants
  • optimiser-friendly
  • requires only one reference to each live object to be discovered (rather than having to find every reference)
Disadvantages of mark-sweep

- Stop/start nature leads to disruptive pauses and long zombie times.

- Complexity is $O(\text{heap})$ rather than $O(\text{live})$
  - every live object is visited in mark phase
  - every object, alive or dead, is visited in sweep phase

- Degrades with residency (heap occupancy)
  - the collector needs headroom in the heap to avoid thrashing
  - Example: lots of marking to do if heap is full and we do this often

- Fragmentation and mark-stack overflow are issues

- Tracing collectors must be able to find roots (unlike reference counting)
  - This needs some understanding of the run-time system or cooperation from the compiler.
Fragmentation: inability to use available memory

• External: allocated memory scattered into blocks; free blocks cannot be coalesced
• Internal: memory manager allocated more space than actually required — common causes are headers, rounding sizes up

Fragmentation is a problem for explicit memory managers as well; `free()` is often not free.
GC is a set-partitioning problem

1. A mark-bit is one way of defining two sets.

2. Mark-compact physically moves members of the live set to a different part of the heap
   - the `free` pointer marks the dividing line between live data and memory that can be overwritten

3. Copying collection is a simpler solution: it picks out live objects and copies them to a ‘fresh’ heap
scan=free
so collection is complete
flip()
    Fromspace, Tospace = Tospace, Fromspace
    top_of_space = Tospace + space_size
    scan = free = Tospace
    for R in Roots {R = copy(R)}
    while scan < free {
        for P in Children(scan) {*P = copy(*P)}
        scan = scan + size (scan)
    }
}

copy(P) {
    if forwarded(P){return forwarding_address(P)}
    else {
        addr = free
        move(P,free)
        free = free + size(P)
        forwarding_address(P) = addr
        return addr
    }
}
Disadvantages of copying GC

- Stop-and-copy may be disruptive
  - Degrades with residency

- Requires twice the address space of other simple collectors
  - touch twice as many pages
  - trade-off against fragmentation

- Cost of copying large objects
  - Long-lived data may be repeatedly copied

- All references must be updated
  - Moving objects may break mutator invariants

- Breadth-first copying may disturb locality patterns
Complexity: caveat emptor

**Claim:** “Copying is always better than mark-sweep GC”

The collectors we've seen so far are very simple minded
Let us compare their basic performance…

Copying is more expensive than setting a bit

*Efficient* implementations of mark-sweep are dominated by cost of mark phase

- linear scanning less expensive than tracing, and
- cost of sweep can be reduced further

Simple asymptotic complexity analyses are misleading
Part 5: Memory management in C++

C++ does not provide automatic memory management

Techniques

RC with smart pointers

Conservative GC using mark-sweep

Finalisation
Important in Destructors in C++:

*Delete all objects in pointer members.*

```cpp
Yclass *p, *q;

... void Xclass() { p = new Yclass(); q = new Yclass();}
void ~Xclass() { delete p; delete q;}
```
"Must do" for pointer members in C++

For all pointer members check:

- Initialisation in *each* constructor
- Deletion in *assignment* operator
- Deletion in *destructor*
- Does copy constructor create shared objects?
- Is *creation* paired with *deletion*?
General advice C++

In general:

• Exploit the concept of ownership
• Check return value of new – it may be null!
• Adhere to convention, e.g., write delete if you write new
• Consider passing and returning objects by value.
• Writing a function that returns a dereferenced pointer is a memory leak waiting to happen!
The **Smart Pointer Concept**

**Basic idea:** allow the programmer to write code that is executed every time a pointer is manipulated:

- Creation
- Assignment
- Copy constructor

**Smart pointers** is a powerful language concept that can be used for many purposes including Garbage Collection.

**The point:** Smart pointers can be thought of as adding a level of indirection: Instead of having a reference to an object, you get a reference to a *smart pointer object* which executes some code every time you use the original reference. The smart pointer object contains a reference to the real object in question.
Common C++ technique:

The basic idea is that the smart pointer object maintains a reference count together with the object reference count.

Template<typename T> class shared_ptr {
    T *ptr;
    long *rc;
public:
    shared_ptr(T* p) : ptr(p) {
        rc = new long;
    }
    ~shared_ptr() { delete ptr; delete rc;}
}
Using Smart Pointers for RC

More smart pointer RC implementation

```cpp
T& operator*() { return *ptr;}
T* operator->() { return ptr;}

shared_ptr& operator=(other object r) {
    if (--*rc == 0) { delete ptr; } // last reference to object
    increment reference count for r
}
```
Smart Pointers comments

Smart pointers are ingenious but their actual implementation is quite gory.

However, they work and can be utilise for many purposes including RC GC.
Finalisation

Finalisers are methods called when an object dies

- explicitly when it is deleted
- implicitly by the collector

Finalisation is commonly used to release scarce resources (e.g. to close files)
Correspondingly, initialisation is allocation.

In non-GC'ed languages, most finalisation is to reclaim memory
- with GC, this action is unnecessary
Simple tracing collectors suffer from several drawbacks
  • disruptive delays
  • repeated work on long-lived objects
  • poor spatial locality

We now outline the approaches taken by sophisticated garbage collectors.
Generational GC

Weak generational hypothesis

“Most objects die young”

It is common for 80-95% objects to die before a further megabyte has been allocated

- 95% of objects are ‘short-lived’ in many Java programs
- 50-90% of CL and 75-95% of Haskell objects die before they are 10kb old
Not a universal panacea

Generational GC is a successful strategy for many but not all programs.

There are common examples of programs that do not obey the weak generational hypothesis.

It is common for programs to retain most objects for a long time and then to release them all at the same time.

Generational GC imposes a cost on the mutator:

- pointer writes become more expensive
Incremental garbage collection

- runs collector in parallel with mutator
- attempts to bound pause time
- many soft real-time solutions
- but no general hard real-time solutions yet
Sequential GC can be made incremental by interleaving collection with allocation.

At each allocation, do a small amount of GC work.

Tune the rate of collection to the rate of allocation to prevent mutator running out of memory before collection is complete.
In conclusion

Points:

• Garbage collection is useful
• You can live without – albeit that can be painful
• Automatic mechanisms for GC are better – even at a slight extra execution time cost
• Conservative collectors actually work
• Classic algorithms reviewed
• There are many advanced collector available
Term: Autumn 2016
Course: CSE 450 (Translation of Programming Languages)
Lead Instructor: Dr. Josh Nahum
TAs:
  Hayam Abdelrahman
  Grant King
You are welcome to fill out a TA SIRS form for one or both TAs.