Memory Management

Ettore Speziale

Politecnico di Milano

Contents

Contents

Why Should Memory be Managed?

Every program uses memory to store data:

Memory Hierarchy

- registers
- caches
- RAM banks
- disks

- registers are fast, small, and expensive
- ...
- disks are slow, huge, and cheap

Language specifications [1] does not expose the structure of the memory hierarchy:

- but inaccurate management of memory leads to using slow levels of memory hierarchy – e.g. swapping to disk

[1] usually

Memory Model

State-of-the-art general purpose languages memory model:

Heap:

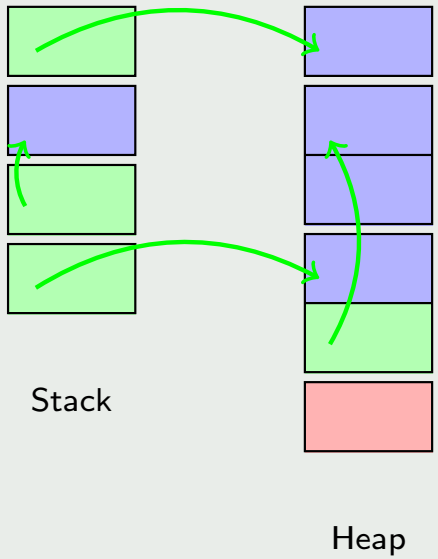
- store stuffs
- manually managed – programmer must explicitly allocates memory on the heap
- partially managed – you can forget deallocating memory, it continues working [a]
- contents not predictable

Stack:

- store global variables
- store function frames; each frame store local variables
- automatically managed – calling a function allocates memory, returning free memory
- possible predicting its contents

[a] in general

Initial Scenario



The red block is not referenced by any pointer:

- it is a leak

Blue blocks store data, while green blocks hold pointers

Where we Are

From your curricula:

C stack is good, heap is a needed evil: use it with care, and always remember to correctly de-allocate memory

Java stack is good, but exploiting its features to increase code quality/performance requires knowing how it works and strictly following a rigid programming discipline: use it only for storing variables for native types, put all other variables on the heap

Post Scriptum:

Java correctly managing the heap is tricky. Just allocate memory, the virtual machine will perform some black-magic to automatically deallocate memory

What Will we See?

This class is about:

- understanding how stack can be used to manage memory
- finding stack-like behaviours, and re-use stack-based techniques to manage the associated memory
- introducing black-magic needed to automate heap management

Contents

Stack Behaviour

Consider the following example:

Stack-based Allocations

```
void foo() {
  Baz baz;
  int bar;
}
```

- storage for bar and baz allocated when foo is called
- storage for bar and baz reclaimed when returning from foo

Let Bar be a class. Stack-allocation induces a regular and predictable behaviour:

- constructor called after memory allocation
- destructor called before memory reclamation

Stack Allocation
Allocating Buffers

Stack-allocated vars can be used to manage memory:

Manual Allocation

```
int foo() {
  int *buf, ret;

  buf = new int[HUGE];

  ...

  delete [] buf;

  return ret;
}
```

Automated Allocation

```
int foo() {
  Buffer<int> buf;
  int ret

  ...

  return ret;
}
```

Automatic Buffer

```
template <typename Ty>
class Buffer {
public:
  Buffer(size_t size = HUGE) :
    buf(new Ty[size]) { }

  ~Buffer() { delete [] buf; }

private:
  Ty *buf;
};
```

Resource Acquisition Is Initialization

The trick used with buffers can be generalize:

RAII a C++ idiom [3]

It exploits C++ scoping rules:

- place RAII object on the stack
- constructor initialize RAII object
- destructor finalize RAII object

Can be applied to other contexts:

Manual Lock

```
void Bar::mutual() {
  lock.acquire();
  lock.release();
}
```

Synch Block

```
void Bar::mutual() {
  Sync sync(lock);
}
```

Owning Pointers

The Buffer<Ty> class has a problem:

- it explicitly de-allocates memory for the buffer

To improve code quality, we need something that:

- acts like a pointer
- manages the life-time of the pointed object
- we call it an owning pointer

For non-array allocations, you can use C++ std :: auto_ptr:

- otherwise, you have to code your owning pointer
  – e.g. llvm :: OwningArrayPtr

Owning Pointers
Buffer

Let std :: auto_arr_ptr an array owning pointer:

Yet Another Automatic Buffer

```
int foo() {
  std::auto_arr_ptr<int> buf;
  int ret

  buf.reset(new int[HUGE]);

  ...

  return ret;
}
```

Reference Counting

Owning pointers can be extended to scenarios when the pointed object is shared:

- an owning pointer cannot be used – it is not safe deleting the object when the owning pointer is destroyed, there can be other pointers referencing it
- we must count the number of times the object is referenced
- when the counter reaches 0, we can delete it
- the mechanism is called reference counting
- the data structure is called shared pointer

Smart Pointers

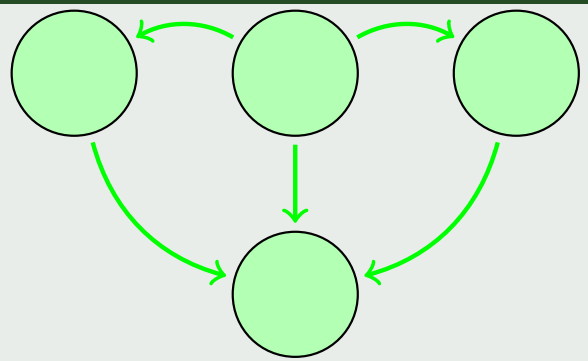
Owning and shared pointers are smart:

- if used correctly, they guarantee automatic and fast memory management

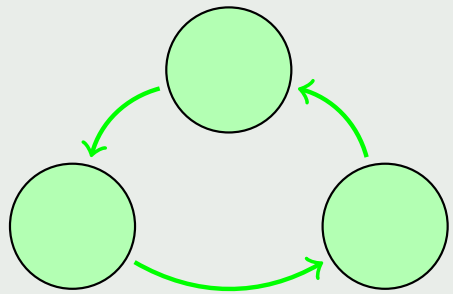
Shared pointers comes with a particular problem:

- loops between objects involving shared pointers must be avoided – impossible detecting that objects in the loop are no more used

No Problems with DAGs



References Loop



Pool Allocation

Sometimes it meaningless allocating new objects:

- e.g. allocating and initializing the object is an expensive operation

In these cases you can exploit pool allocation:

- for each type, keep a set of objects of that type – a pool
- when a new instance of type foo is needed, look in the pool
- if not empty, allocate a new object by removing it from the pool
- otherwise, create a new object
- in any case, when deleting the object return it into the pool

Pool Allocation
Thread Pool

Typical example of pool is the thread pool:

- creating threads is expensive
- *cache* them using an allocation pool

Thread Pool

```
class Thread {
public:
  static Thread &get();
  static void put(Thread &thr);

private:
  static std::vector<Thread *> pool;
};
```

Thread instances managed through factory functions:

Thread Pool Usage

```
Thread &thr = Thread::get();
...
Thread::put(thr);
```

Of course, Thread constructors must be **private**:

- Thread instaces created only using factory functions

Implementation is trivial:

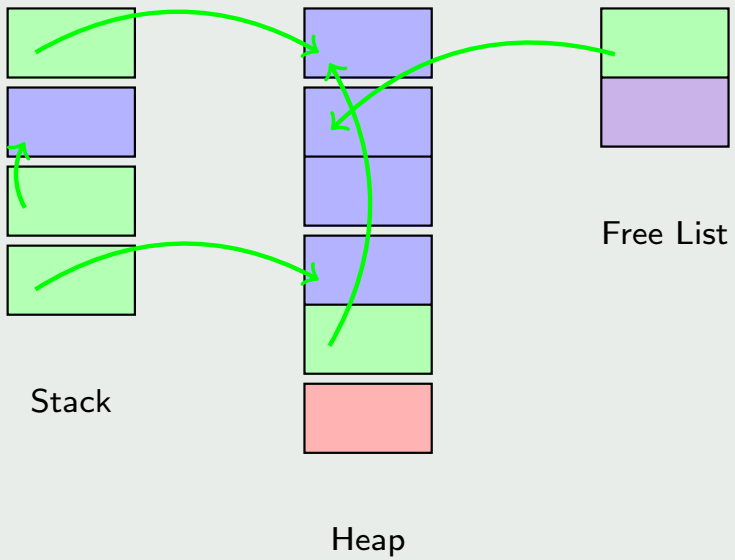
Creating Threads

```
Thread &Thread::get() {
  if(pool.empty()) {
    return *(new Thread());

  Thread *thr = pool.back();
  pool.pop_back();

  return *thr;
}
```

Heap allocation requires finding a free block of memory with a given size:

- the allocator simple keeps a list of free blocks – free list
- let n be the size of such list: allocation cost is $\mathcal{O}(n)$

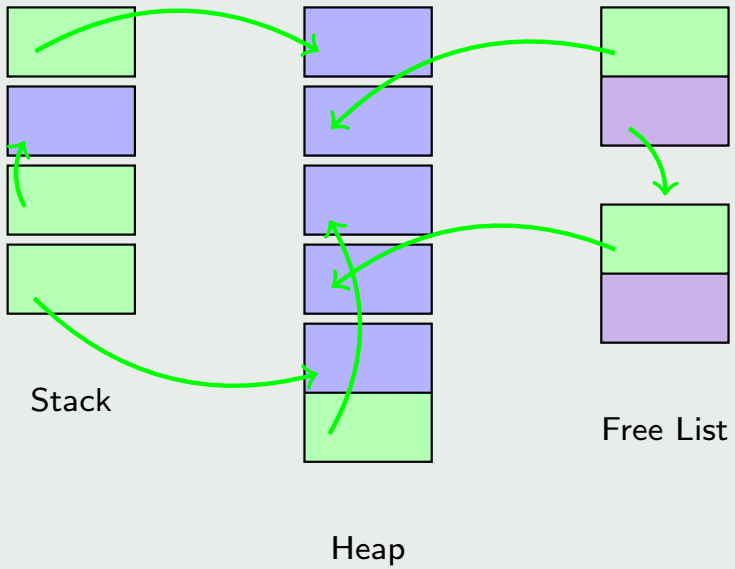
Heap as Seen by **new**



Dellocation requires returning a block back to the free list:

- heap can be fragmented
- once in a while it must be de-fragmented – e.g. every n allocations

Fragmented Heap

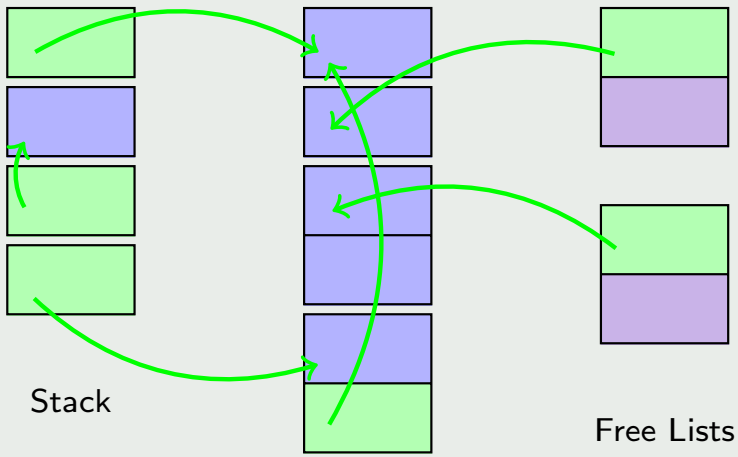


Memory Pool
Improving Performance

We can improve alloocation performance by:

- consider a small set of possible size ranges
- keep a free list for each range

Multiple Free Lists



Faster than before, but:

- alloc time is $\mathcal{O}(n)$

Optimization:

- use pool allocation to manage frequently instanced types
- alloc time is $\mathcal{O}(1)$

Parametric Pooled Allocator

```
template <typename Ty>
class Allocator {
public:
  static void *operator new(size_t size);
  static void operator delete(void *addr);

private:
  static std::vector<Ty *> pool;
};
```

Overriding **operator new** and **operator delete** allows using standard interface for allocating objects

Thread-based Allocators

What about threads?

- free lists/allocators are shared resources
- you need to synchronize threads while accessing them
- but synchronization is expensive

Solution is to avoid synchronization:

- each thread has a private allocator
- no synchronization at alloc-time
- synchronization at free-time needed only when current thread is different from the one who has allocated the block

Contents

What is Garbage Collection?

Garbage collection is a technique used to manage the heap:

- it simplify heap management by avoiding programmers explicitly freeing memory blocks once they becomes unnecessary
- unused blocks are automatically identified and freed

Reference counting is a simple garbage collector:

- counts references to memory blocks – e.g. objects
- when counter reaches 0, free used memory

Garbage collector usually used with high-level languages:

- its efficiency depends on language-specific support
- C/C++ lacks needed data-structures, thus not very efficient with these languages

Starting Point

Let us consider the following problem:

- we want to implement a garbage collector
- we can allocate blocks on the heap
- we have to find a way to identify unused blocks and free them

Reference counting is not a general technique:

- unable to detect loops between blocks

However, we can exploit a similar concept:

- we must observe pointers

Tracing Collectors

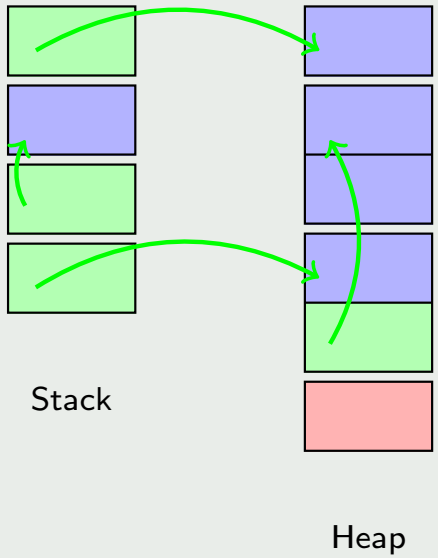
A tracing garbage collector assumes blocks are the nodes of a graph:

- an edge from node foo to node bar means that inside block foo there is a pointer referencing node bar
- on the graph there are some special nodes. They represent pointers on stack referencing objects in the heap
- these pointers constitutes the root set

Tracing collection is simple:

- visit the graph starting from nodes in the root set
- after graph traversal, unvisited blocks represent blocks no more referenced by the application – free them

Heap and Stack in Tracing Collectors



Root set contains two pointers:

- stack-allocated pointer referencing stack-allocated data is not considered by garbage collector

Red block not allocated by garbage collector:

- not considered for collection

Tracing Collectors
The Challenging Part

The big problem of tracing collectors is detecting the root set and pointers inside blocks:

managed languages you cannot use pointers, you have to use references [2]. The compiler/runtime knows locations of references, so the garbage collector knows where pointers are – precise garbage collection – e.g. Java, LISP

unmanaged languages you can use pointers. The contents of heap-allocated blocks are unknown. The root set is unknown. Both problems derive from a weak type system and usage of pointer arithmetics – conservative garbage collection – e.g. C, C++

[2] Pointers without arithmetic

Conservative Garbage Collectors
Finding Pointers

With unmanaged languages, pointers must be estimated:

- the garbage collectors knows the range of addresses forming the heap
- it also knows stack base and top

Root set can be estimated as follows:

- interpret the stack as an array of pointers
- walk the whole array
- if an element of the array points to something falling inside garbage collector-managed heap, put it into the root set

The same is done for each block inside the heap:

- interpret the block as an array of pointers
- if an element of the array points to something falling inside the heap, it is a successor of the block

Visiting the Heap

Knowing the root set and how detecting pointers inside heap blocks, we can visit the heap looking for garbage:

- there are three basic techniques
- they differentiate on how heap is managed

Mark and Sweep

Start from the root set:

- visit the graph
- mark each visited node

Then:

- scan all blocks in the heap
- if a block is not marked, it is garbage – free it

Collection is fast and simple, but:

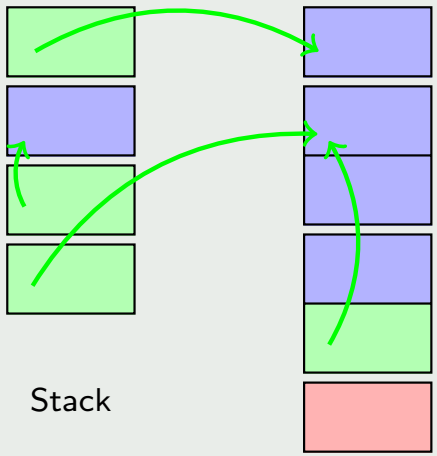
- induces heap fragmentation
- free lists are needed to keep track of free blocks

Mark and Sweep
Example

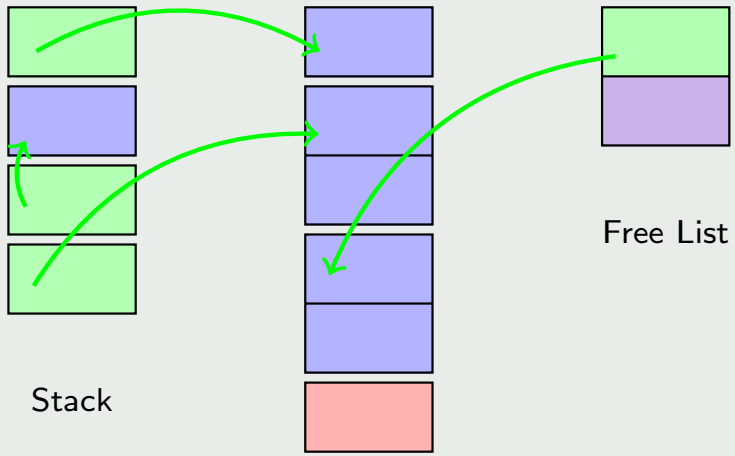
Free list is empty before collection:

- no blocks available for allocations

Before Collection



After Collection



After collection free list holds pointer to available blocks

Mark and Compact

Goal is to improve mark and sweep performance by de-fragmenting the heap during the sweep phase [3]:

- scan all blocks in the heap
- if a block is not marked, it is garbage – next block can be shifted, in order to fill the slot occupied by the unmarked block
- at the end of the collection, all marked object are compacted on the bottom of the heap – free space starts after last marked object

[3] Mark phase is unchanged

Mark and Compact
Notes about Compaction

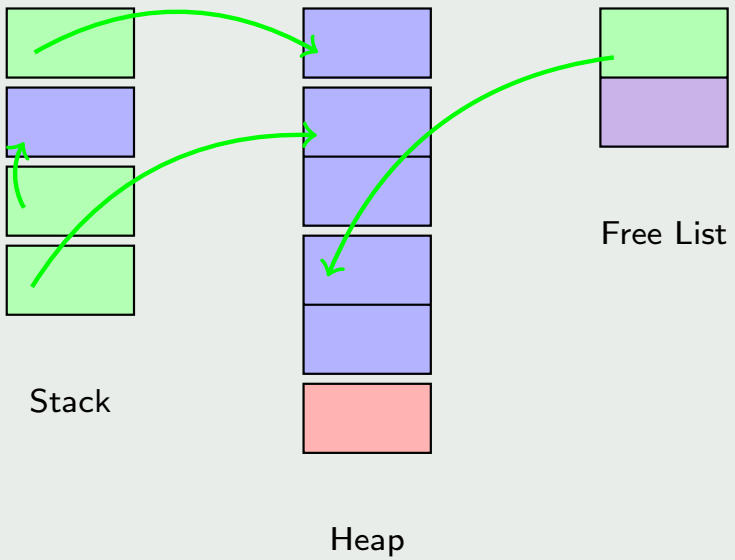
Mark and compact moves blocks:

- pointers to blocks must be updated – pointer reversal
- it requires precise information about pointers – usable only with managed languages
- multiple sweep phases can be executed – e.g. finding unmarked objects, computing deltas in freed space, update pointers to marked objects, compact marked objects

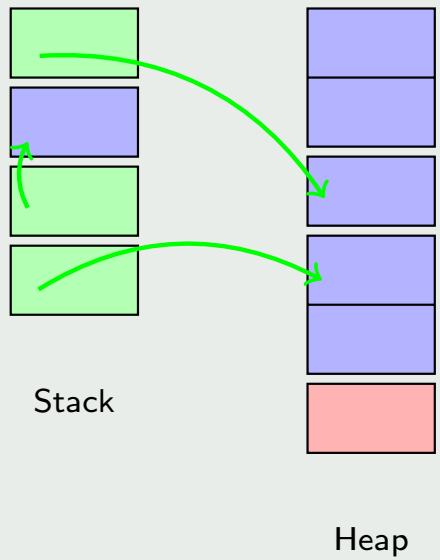
Mark and Compact
Example

Mark and sweep needs a free list because free blocks can be anywhere in the heap

After Mark and Sweep



After Mark and Compact



Mark and compact can avoid using a free list:

- free blocks are always on top of the heap

Copying Collectors

Copying collectors partition the heap into two segments:

from-space holds reachable blocks. Allocation of new blocks are performed here – all blocks in the graph are stored here

to-space a support space, used during collections

As usual, collection starts from the root set:

- for all visited block, move it from the from-space to the to-space
- after moving a block, overwrite old location with a mark, recording block new address in the to-space
- if a block has already been visited, its new address is found in the from-space – use it to perform pointer-reversal

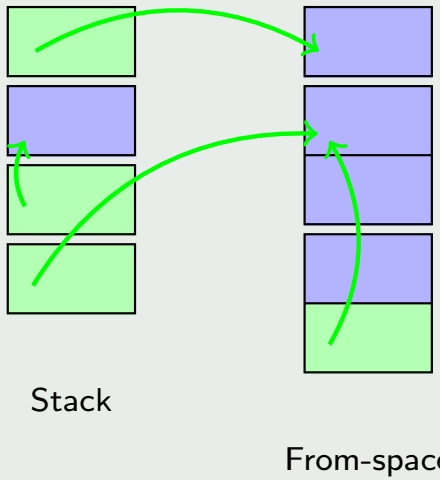
At the end of the collection, swap from-space and to-space

Copying Collectors
Comment

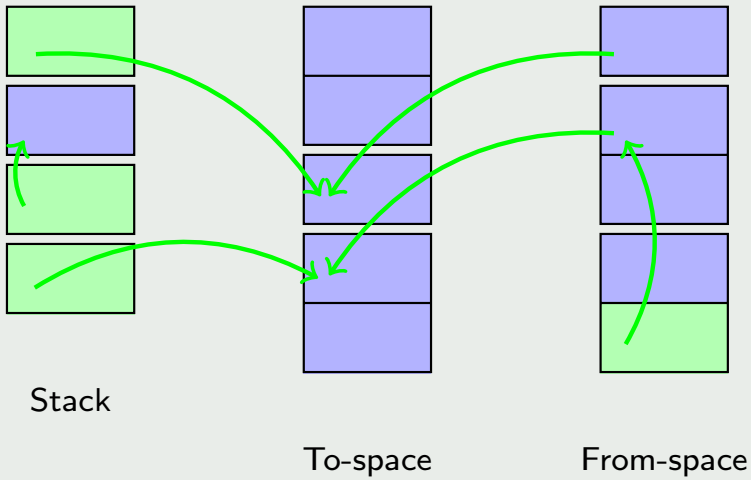
Copying collectors visit blocks only one time:

- no need of multiple sweeps to correctly update pointers
- pointers are updated while walking the graph

Before Collection



Just Before Swapping Spaces



When Should Memory be Collected?

The ideal collection instant is when the number of reachable block is minimum:

- time spent in graph traversal is minimized
- freed memory is maximized

However, detecting this instant is not easy:

- memory should be observed, but this requires time and since this preventes using the same time for running user program, garbage collectors play statistic – e.g. start collection when free space is around 30% of heap size
- basic algorithms execute collection inside memory allocation routines – e.g. a block is needed, and free space is below the threshold: trigger collection

Contents

Real-world Memory Allocation

For every-day applications:

- malloc/**new** provides acceptable performances

For high-performance, allocation-intensive applications:

- use a custom allocator, in order to optimize heap management
- advanced libraries – e.g. C++ STL – allows to configure the internally used memory allocator

Real-world Garbage Collection

We have seen the basic algorithms:

- production-quality garbage collectors partition heaps in section, each manages by a basic algorithm
- state-of-the-art collectors are BDW [1] and G1 [2]

Please remember that:

- garbage collection simplify memory management, but its overhead is non-negligible
- it is good for managing unstructured allocations
- if the allocation pattern exposes some kind of regularity – e.g. think about tree nodes –, owning/shared pointers simplify memory management without incurring into relevant performance penalties

Contents

Bibliography

Hans Boehm.
A Garbage Collector for C and C++.
http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html.

David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis.
Garbage-first Garbage Collection.
In *ISMM*, pages 37–48, 2004.

Wikipedia.
C++ Programming/RAII.
http://en.wikibooks.org/wiki/C++_Programming/RAII.

Paul R. Wilson.
Uniprocessor Garbage Collection Techniques.
In *IWMM*, pages 1–42, 1992.