Compiler Design

Lecture 21: Garbage Collection

Christophe Dubach
Winter 2022

Original slides by Prof. Laurie Hendren,
updated by Alex Krolik, 2016 – 2020,
updated by Christophe Dubach, 2021 – 2022.

Timestamp: 2022/03/31 16:39:00

Garbage Collection

Memory Management

Manual Deallocation Mechanisms

Runtime Automatic Deallocation Mechanisms

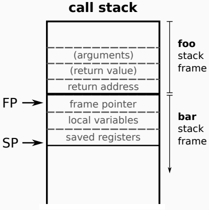
- Reference Counting
- Mark-and-sweep
- Stop-and-Copy
- Pratical Considerations

Memory Allocation

Stack Memory Allocation

- Space allocated in the function call stack;
- Is used for function call information, local variables, and return values;
- Typically contains fixed size data; and
- Is allocated and deallocated at the beginning and end of a function.

Information stored in the stack is therefore specific to a particular function invocation (*i.e.* call).

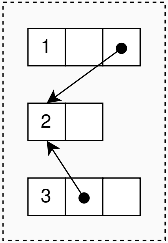


Heap Memory Allocation

- Space allocated in the program heap;
- Is very dynamic in nature:
  - Unknown size; and
  - Unknown time;
- Requires additional runtime support for managing the heap space.

Information stored in the heap is therefore not necessarily tied to any particular function invocation.

Example Heap



Heap variables may be referred to by other objects in the heap, or from the stack.

Heap Memory Allocation

Data stored in the heap is controlled by a heap allocator (*i.e.* `malloc`).

- Manages the memory in the heap space;
- Takes as input an integer representing the size needed for the allocation;
- Finds unallocated space in the heap large enough to accommodate the request; and
- Returns a pointer to the newly allocated space.

You will find more details in an operating systems course.

Heap Memory Deallocations

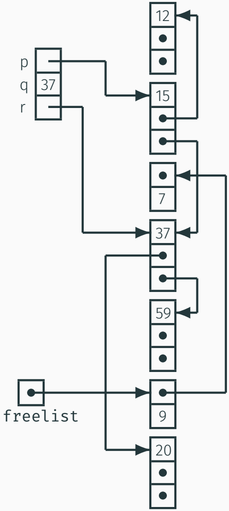
Memory allocated on the heap is freed when it is no longer needed. This can be:

- **Manual**:
    - User code makes the necessary decisions on what is live and when to deallocate (*e.g.* using `free`).
- **Automatic**:
    - Continuous: Runtime code determines on the spot which objects are live; or
    - Periodic: Runtime code determines at specific times which objects are live.

Without runtime support the *program* must explicitly return the memory when it is no longer needed.

Heap Memory Deallocations

For this class, we will assume that the freed heap blocks are stored on a `freelist` (a linked list of heap blocks). Freeing an object *prepends* the heap block onto the list.



Garbage Collection

Memory Management

Manual Deallocation Mechanisms

Runtime Automatic Deallocation Mechanisms

- Reference Counting
- Mark-and-sweep
- Stop-and-Copy
- Pratical Considerations

Manual Deallocation Mechanisms

Heap memory can be freed manually at any point in the program.

- Programmers determine when an object is no longer live; and
- Requires calls to a deallocator (i.e. `free`).

Consider the following code

```
int *a = malloc(sizeof(int));

// ... use of a

free(a);

...

*a = 5; // what happens?
```

Manual Deallocation Mechanisms

Advantages

- Reduces runtime complexity;
- Gives the programmer full control on what is live; and
- Can be more efficient in some circumstances.

Disadvantages

- Requires extensive effort from the programmer;
- Gives the programmer full control on what is live;
- Error-prone; and
- Can be less efficient in some circumstances.

Manual Deallocation Mechanisms

Sometimes manual deallocation is slower than automatic methods. Consider the following example code, which allocates 100 integers and then deallocates them one-by-one.

```
for (int i = 0; i < 100; ++i) {
    a[i] = malloc(sizeof(int));
}

...

for (int i = 0; i < 100; ++i) {
    free(a[i]);
}
```

This is potentially inefficient. Why?

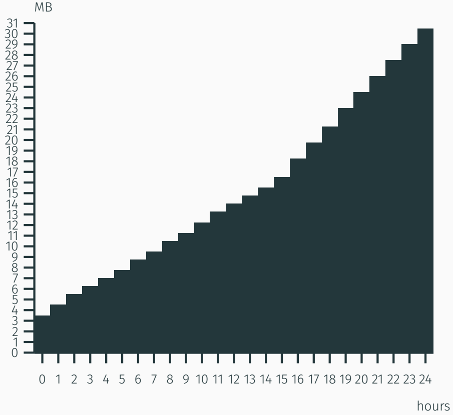
The allocations are potentially contiguous, and could therefore be reclaimed as a *block* instead of one-by-one.

Manual Deallocation Mechanisms

Life Without Garbage Collection

- Dead records must be explicitly deallocated;
- “Superior” if done correctly; but
- It is easy to miss some records;
- It is “dangerous” to handle pointers; and
- May be less efficient in some cases.

Memory leaks in real life (`ical v.2.1`)



Garbage Collection

Memory Management

Manual Deallocation Mechanisms

Runtime Automatic Deallocation Mechanisms

- Reference Counting
- Mark-and-sweep
- Stop-and-Copy
- Pratical Considerations

Runtime Deallocation Mechanisms

A runtime deallocation mechanism must answer the question:

Which records are *dead*, *i.e.* no longer in use?

The more precise the answer, the better the deallocation mechanism.

Ideally: Records that will never be accessed in the future.

→ But this is undecidable, *e.g.*:

```
int * foo(int a) {
    int *p1 = (int*)malloc(16);
    int *p2 = (int*)malloc(16);
    if (a<3) return p1 else return p2;}
```

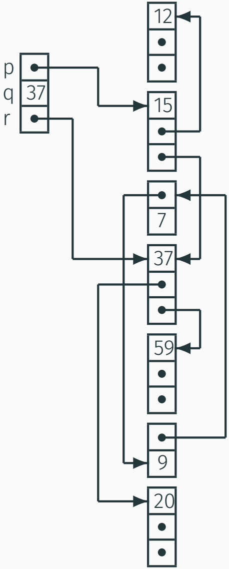
Basic conservative assumption

- A record is *live* if it is reachable from a stack-based program variable or global variable, otherwise dead.

Example Heap

Consider the following example heap, with stack variables: `p`, `q` and `r`.

- Which records are live?
- Which records are dead?



Runtime Deallocation Mechanisms

A *garbage collector*

- Is part of the runtime system; and
- Automatically reclaims heap-allocated records that are no longer used.

A garbage collector should

- Reclaim *all* unused records;
- Spend very little time per record;
- Not cause significant delays; and
- Allow all of memory to be used.

These are difficult and often conflicting requirements.

Garbage Collection

In this class we will study three types of garbage collection:

- Reference counting;
- Mark-and-sweep; and
- Stop-and-copy.

For each algorithm we will discuss the implementation, an example, and the associated advantages/disadvantages.

Runtime Automatic Deallocation Mechanisms

Reference Counting

Reference Counting

- Is a type of continuous (or incremental) garbage collection;
- Uses a field on each object (the reference count) to track incoming pointers; and
- Determines an object is dead when its reference count reaches zero.

The reference count is updated

- Whenever a reference is changed:
    - Created
      e.g. `int *a = b; // b refcount++`
    - Destroyed
      e.g. `a = c; // b refcount--`
- Whenever a local variable goes out of scope;
- Whenever an object is deallocated (all objects it points to have their reference counts decremented).

Reference Counting

Reference counting inserts calls to `Increment` and `Decrement` in the source program as needed. When the object is no longer needed, the call to `Free` is made.

Pseudo code for reference counting

```
function Increment($x$)
    $x$.count := $x$.count+1

function Decrement($x$)
    $x$.count := $x$.count−1
    if $x$.count = 0 then
        Free($x$)
```

```
function Free($x$)
    for $i$:=1 to $|x|$ do
        Decrement($f_i$)
    $x.f_1$ := freelist
    freelist := $x$
```

Example

```
class A { int count; // ref counting
          int y;}
class B { int count; // ref counting
          int x;
          A a;}
B foo(int c) {
    A a1 = new A();
    A a2 = new A();
    B b  = new B();

    b.a = a1;



    if (c>4) {
        b.a = a2;


    }



    return b;
}
```

Example with increment/decrement calls

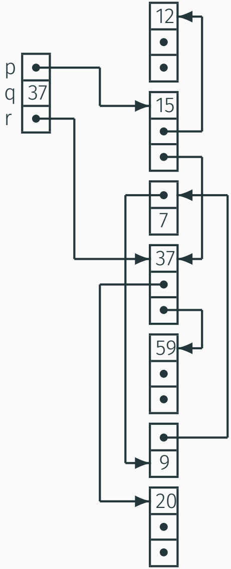
```
class A { int count; // ref counting
          int y;}
class B { int count; // ref counting
          int x;
          A a;}
B foo(int c) {
    A a1 = new A(); // a1.count = 1;
    A a2 = new A(); // a2.count = 1;
    B b  = new B(); // b.count = 1;

    b.a = a1;
    decrement(b.a); // assuming b.a is null, nothing to do
    increment(a1);  // a1.count = 2;

    if (c>4) {
        b.a = a2;
        decrement(b.a); // a1.count = 1;
        increment(a2);  // a2.count = 2;
    }

    decrement(a1);  // a1.count = ...
    decrement(a2);  // a2.count = ...
    return b;
}
```

Reference Counting



Reference counting has one big problem:

- What about objects 7 and 9?

Reference Counting

Advantages

- Is incremental, distributing the cost over a long period;
- Does not require long pauses to handle deallocations;
- Catches dead objects immediately; and
- Requires no effort from the user.

Disadvantages

- Is incremental, slowing down the program continuously; and
- Cannot handle circular data structures.

Automatic Reference Counting (ARC)

Initially for Objective-C (now also for Swift), *automatic reference counting* (ARC) is a reference counting implementation designed by Apple and integrated into Clang.

- Inserts calls to `retain` (increment) and `release` (derement) at compile time;

Previously, developers inserted manual calls to the memory management methods.

Smart pointers

Alternative to ARC is to use smart pointers, such as ones found in the C++ Standard Library

- `unique_ptr`, `shared_ptr`, `weak_ptr`
- automatically decremented when becomes out of scope (using C++ destructor mechanism)
- if you want to learn more: https://en.cppreference.com/book/intro/smart_pointers

Runtime Automatic Deallocation Mechanisms

Mark-and-sweep

Mark-and-Sweep

The mark-and-sweep algorithm is a periodic approach to garbage collection that has 3 main steps:

1. Explore pointers starting from the program (stack) variables, and *mark* all records encountered;
2. *Sweep* through all records in the heap and reclaim the unmarked ones; and
3. Finish by unmarking all marked records.

Assumptions

- We know which fields are pointers;
- We know the size of each record; and
- Reclaimed records are kept in a `freelist`.

Mark-and-Sweep

The 3 steps of the mark-and-sweep algorithm are shown below (steps 2 and 3 are merged).

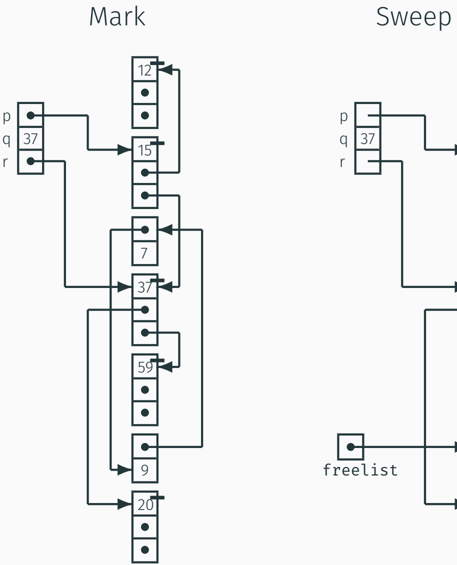
Pseudo code for mark-and-sweep

```
function Mark()
    for each program variable $v$ do
        DFS($v$)

function DFS($x$)
    if $x$ is pointer into heap then
        if record $x$ not marked then
            mark record $x$
            for $i$:=1 to $|x|$ do
                DFS($x.f_i$)
```

```
function Sweep()
    $p$ := first address in heap
    while $p <$ last address in heap do
        if record $p$ is marked then
            unmark record $p$
        else
            $p.f_1$ := freelist
            freelist := $p$
        $p$ := $p$+sizeof(record $p$)
```

Mark-and-Sweep



Analysis of Mark-and-Sweep

- Assume the heap has size H words; and
- Assume that R words are reachable.

The cost of garbage collection: $\mathcal{O}(R + C)$

Mark-and-Sweep

Advantages

- Is periodic, so does not slow down each operation in your program;
- Can be run in parallel to your program;
- Mark and sweep steps can be parallelized too;
- Requires no effort from the user.

Disadvantages

- Scanning the heap can be expensive;
- The heap may become *fragmented*: containing many small free records but none that are large enough for the next allocation.

Heap Fragmentation

To deal with fragmented heaps we can use *compaction*.

- Once mark-and-sweep has finished, collect all live objects at the beginning of the heap;
- Adjust pointers pointing to all moved objects;
- The adjustment depends on the amount of space freed before the object;
- This removes fragmentation and improves locality.

This is not possible in all programming languages as garbage collection must be conservative.

$\Rightarrow$ How do we know which fields are pointers?

Runtime Automatic Deallocation Mechanisms

Stop-and-Copy

Stop-and-Copy

Stop-and-copy is a periodic approach to garbage collection that:

- Divides the heap into two parts;
- Only uses one part at a time;

Conceptually this results in a simple high-level algorithm:

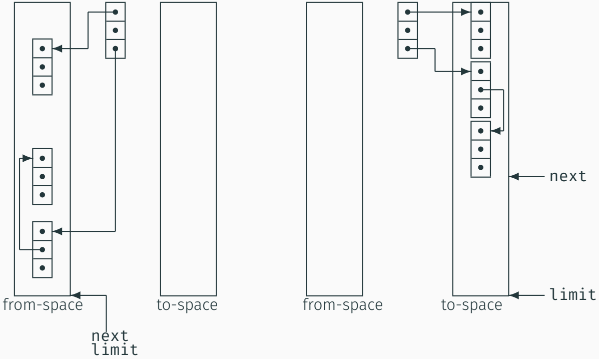
1. Use the active half of the heap for all allocations;
2. When it runs full, copy live records to the other part; and
3. Switch the roles of the two parts.

A Nonrecursive List Compacting Algorithm, C. J. Cheney, Commun. ACM, 1970.

Stop-and-Copy

Consider the following snapshots of stop-and-copy before/after execution.

- `next` and `limit` indicate the available heap space; and
- Copied records are contiguous in memory.



Stop-and-Copy

Objects on the stack are processed one at a time as follows:

- If the object has not yet been moved to to-space, copy the object fields to to-space and update the first field of the from-space version with a *forwarding* reference to the to-space copy. Then update the objet reference from the stack to refer to the copy in the to-space.
- If the object has already been moved to to-space, update the object reference from the stack to refer to the copy in the to-space (using first field of the object which points to the copy).

Then, we simply process all the objects in the to-space, performing one of the actions above for each field of the object (which holds a reference).

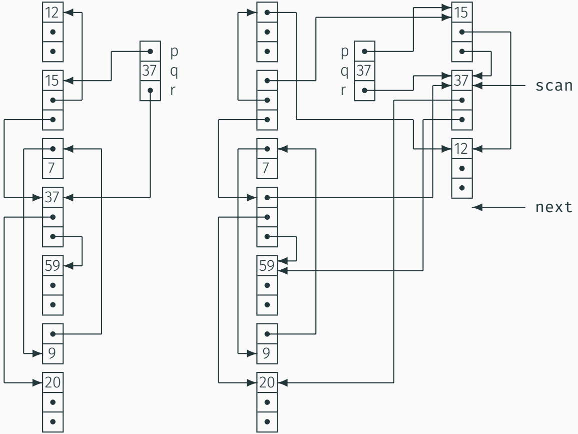
Pseudo code for stop-and-copy

```
function Copy()
  scan := next := start of to-space
  for each object $obj$ on stack do
    $obj$ := Forward($obj$)
  while scan $<$ next do
    for $i$:=1 to |scan| do
      scan.$f_i$ := Forward(scan.$f_i$)
    scan := scan+sizeof(record scan)
```

```
function Forward($obj$)
  if $obj \in$ from-space then
    if $obj.f_1 \in$ to-space then
      // copy already made
      return $obj.f_1$
    else // copy obj to new space
      for $i$:=1 to $|obj|$ do
        next.$f_i$ := $obj.f_i$
      $obj.f_1$ := next
      next := next+sizeof($obj$)
      return $obj.f_1$
  else return $obj$
```

Stop-and-Copy

The follow are snapshots of stop-and-copy before executing and after forwarding the top-level and scanning 1 record.



before

after forwarding p, q, and r and scanning 1 record

Analysis of Stop-and-Copy

- Assume the heap has size H words; and
- Assume that R words are reachable.

The cost of garbage collection is now: $\mathcal{O}(R)$

Stop-and-Copy

Advantages

- Allows fast allocation (no `freelist`);
- Avoids fragmentation;
- Collects in time proportional to R.

Disadvantage

- Wastes half your memory; and
- Stops the program to execute.

Runtime Automatic Deallocation Mechanisms

Pratical Considerations

Practical Considerations

In practice, we use either mark-and-sweep or stop-and-copy (and in some systems ref. counting).

This can lead to better memory management, at the cost of ~ 100 instructions for a small object.

Each algorithm can be further extended by

- Generational collection (to make it run faster); and
- Incremental (or concurrent) collection (to make it run smoother).

Generational Collection

Observation: the young die quickly

Given this assumption, the garbage collector should:

- Focus on young records;
- Divide the heap into generations: $G_0, G_1, G_2, \ldots$;
- All records in G_i are younger than records in G_{i+1};
- Collect G_0 often, G_1 less often, and so on; and
- Promote a record from G_i to G_{i+1} when it survives several collections.

Other Optimizations

Exploit locality

- Keep youngest generation small enough to fit in the CPU cache

Incremental collection

- Garbage collection may cause long pauses;
- This is undesirable for interactive or real-time programs; so
- Try to interleave the garbage collection with the program execution (*e.g. Train algorithm*).

Earlier Assumptions

The presented garbage collection algorithms assumed that:

- We know the size of each record; and
- We know which fields are pointers.

For object-oriented languages, each record already contains a pointer to a class descriptor, so garbage collection is straightforward to implement.

For general languages, we must sacrifice a few bytes per record to indicate its size, and its organization.