Garbage Collection

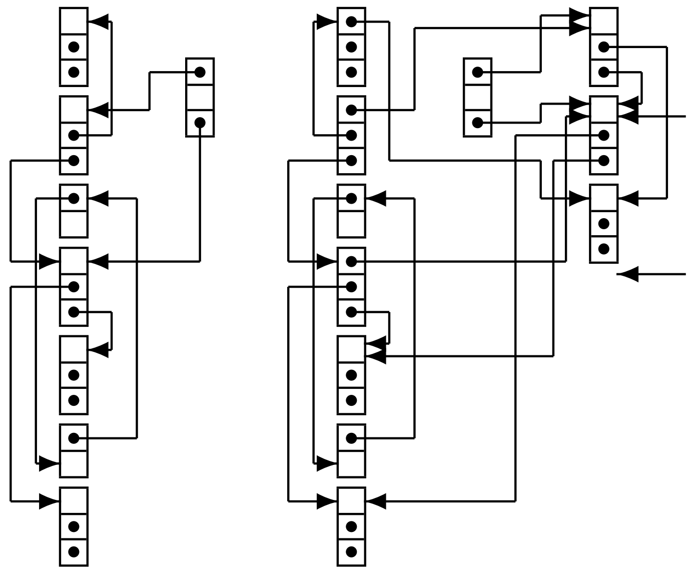
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/



Compiley "Mompiler" McCompilerface

Announcements (Monday, March 11th/Wednesday, March 13th)

Milestones

- Milestone 1 graded (programs on myCourses)
- Milestone 2 due: Sunday, March 17th 11:59 PM
- Peephole due: Friday, April 12th 11:59 PM

To come

- Milestone 3 out: Monday, March 18th. Due: Wednesday, March 27th 11:59 PM
- Milestone 4 out: Monday, March 18th. Due: Wednesday, April 10th 11:59 PM
- Final report out: Monday, March 18th. Due: Wednesday, April 10th 11:59 PM
- Group meeting: Week of April 8th (you may request an extension until the week of April 15th)
- Final exam: Thursday, April 18th 2:00 PM

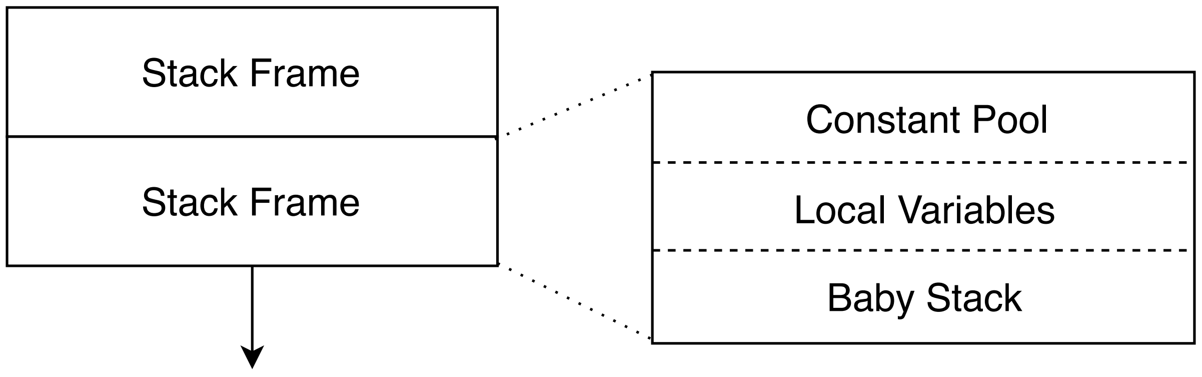
Memory Allocation

Stack Memory Allocation

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

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

Example Stack



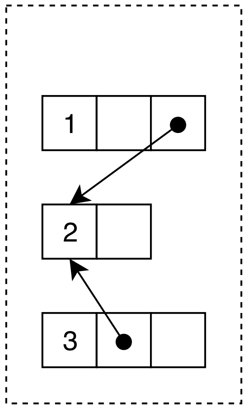
Memory Allocation

Heap Memory Allocation

- Allows space to be allocated and deallocated as needed and in any order;
- 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

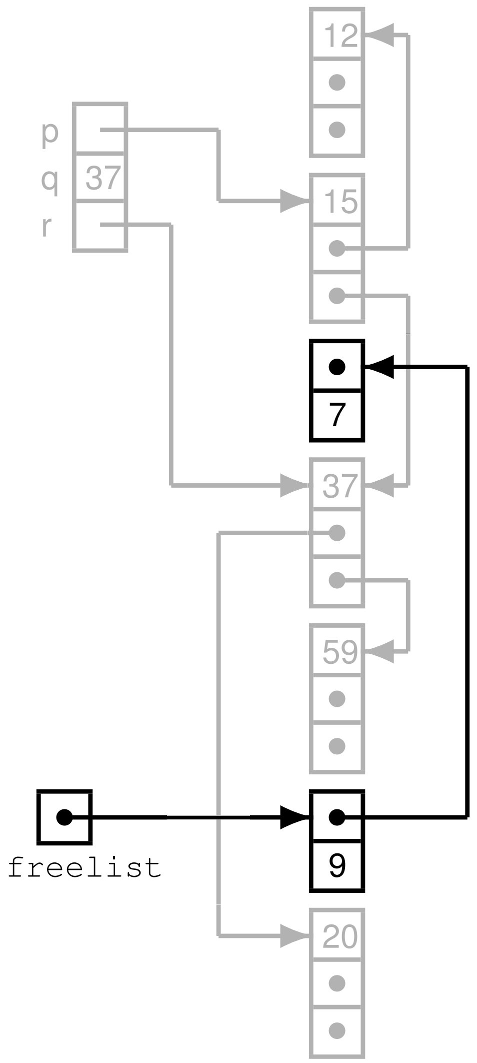
Memory allocated on the heap are freed when they are no longer live (i.e. `free`). This can be

- Manual: User code making the necessary decisions on what is live;
- Continuous: Runtime code determining on the spot which objects are live; or
- Periodic: Runtime code determining at specific times which objects are live.

Without runtime support it is up to the *program* to 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)



Manual Deallocation Mechanisms

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

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

Consider the following code

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

[...]

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

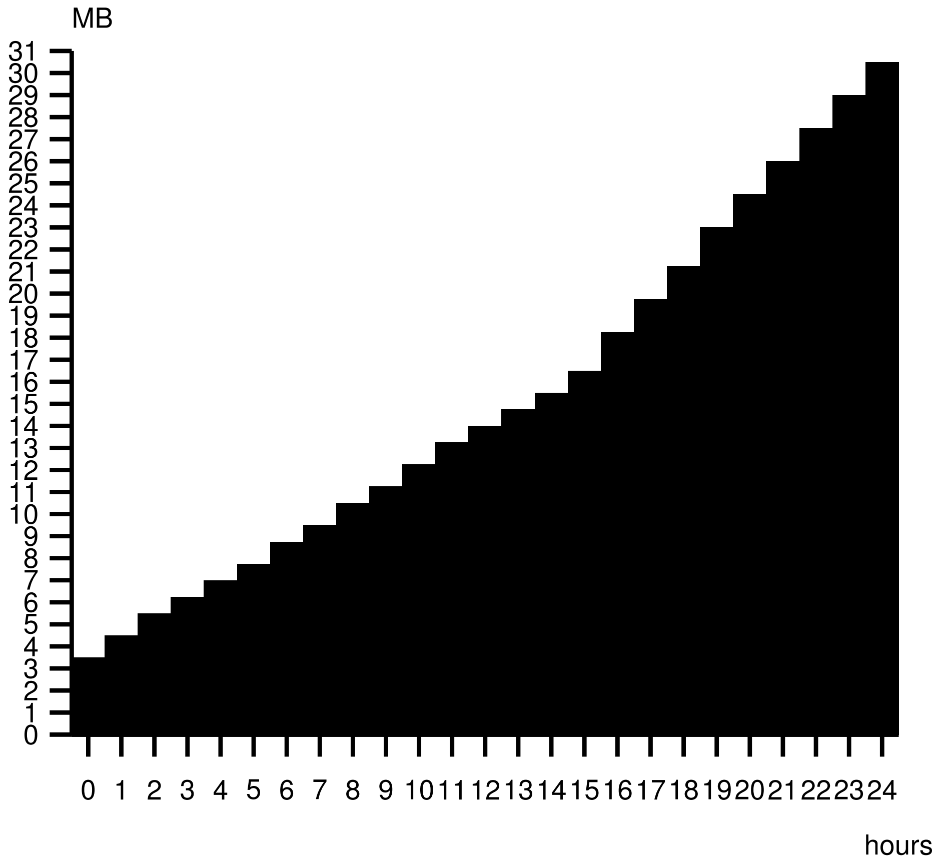
```
for (int i = 0; i < 100; ++i)
{
   a[i] = malloc(sizeof(int));
}

[...]

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

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

Life Without Garbage Collection

- Unused records must be explicitly deallocated;
- “Superior” if done correctly; but
- It is easy to miss some records; and
- It is “dangerous” to handle pointers.

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



Runtime Deallocation Mechanisms

A *garbage collector*

- Is part of the runtime system;
- 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.

Runtime Deallocation Mechanisms

Any correct runtime deallocation must answer the fundamental question

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

The more precise the answer, the better the garbage collector.

Ideally

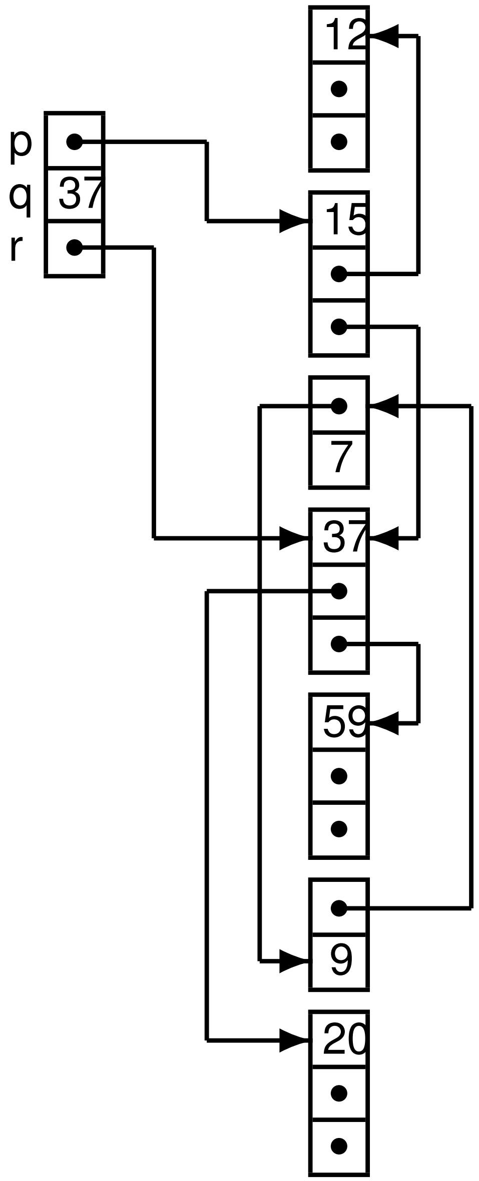
- Records that will never be accessed in the future execution of the program; but
- This is undecidable.

Basic conservative assumption

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

Note: Dead records may still be pointed to by other dead records.

Example Heap



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

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

Garbage Collection

In this class we will study 3 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 advantages/disadvantages.

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

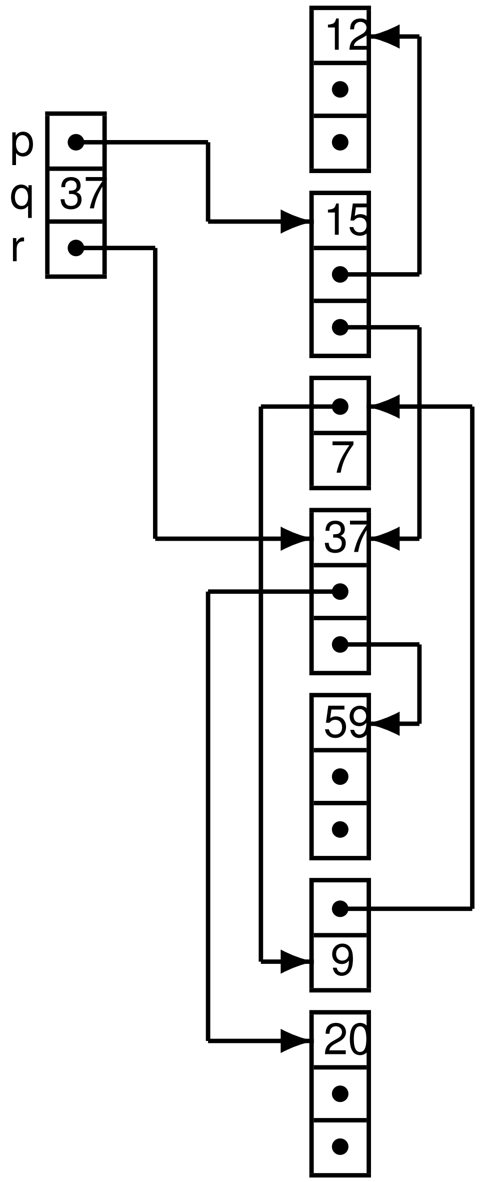
Reference counting inserts calls to `Increment` and `Decrement` in the source program as needed. When the object is no longer needed, the call to `Free` is made.

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($x.f_i$)
    $x.f_1$ := freelist
    freelist := $x$
```

Reference Counting



Reference counting has one large problem:

What about objects 7 and 9?

Reference Counting

Advantages

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

Disadvantages

- Is incremental, slowing down the program continuously and unnecessarily;
- Requires a more complex runtime system; and
- Cannot handle circular data structures.

Aside: 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` at compile time;
- **Optimizes away unnecessary updates**; and
- Is preferred to garbage collection.

Previously, developers inserted calls to the memory management methods.

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

Mark-and-Sweep

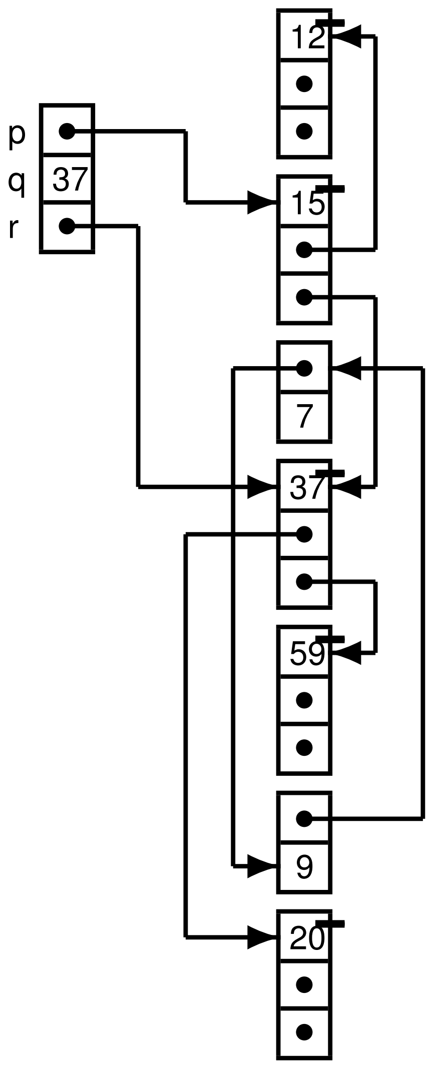
The 3 steps of the mark-and-sweep algorithm are shown below (steps 2 and 3 are merged)

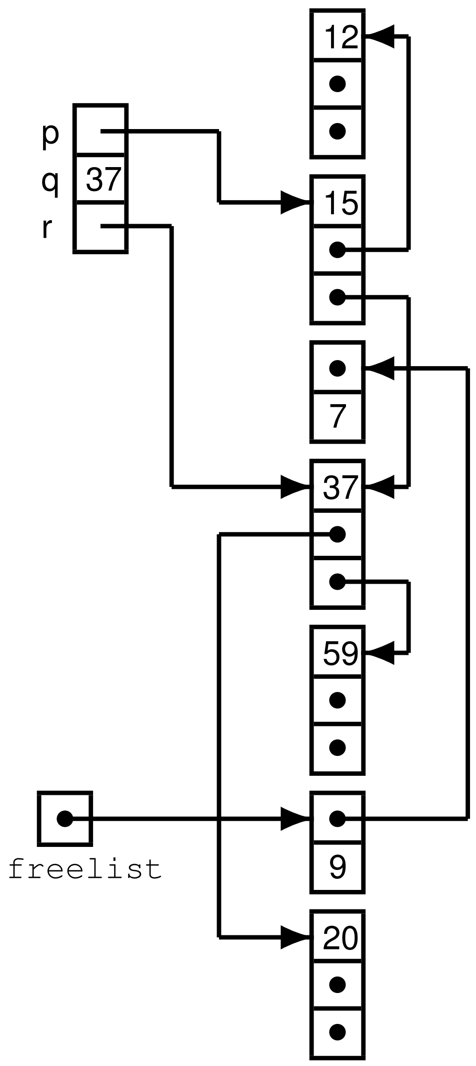
Pseudo code for mark-and-sweep

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

function DFS($x$)
    if $x$ is a pointer into the heap then
        if record $x$ is 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

$$c_1 R + c_2 H$$

Realistic values

$$10R + 3H$$

The cost per reclaimed word

$$\frac{c_1 R + c_2 H}{H - R}$$

- If R is close to H, then this is expensive;
- The lower bound is c_2;
- Increase the heap when $R > 0.5H$; then
- The cost per word is $c_1 + 2c_2 \approx 16$.

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.

Heap Fragmentation

To deal with fragmented heaps we 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 due to the conservative nature of garbage collection.

DFS Recursion Stack

For mark-and-sweep, the recursion stack could have size H (and has at least size $\log H$). However, given that we have (potentially) run out of memory, it may be impossible to allocate!

Pointer reversal

Pointer reversal is a clever method for traversing a tree, that embeds the stack in the fields themselves.

Idea: Instead of recursively calling DFS, temporarily update the field in the record to point to the *parent* node in the traversal.

`http://www.cs.cmu.edu/afs/cs/academic/class/15745-s06/web/handouts/garbage.pdf`

Stop-and-Copy

The stop-and-copy algorithm is a periodic approach to garbage collection that

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

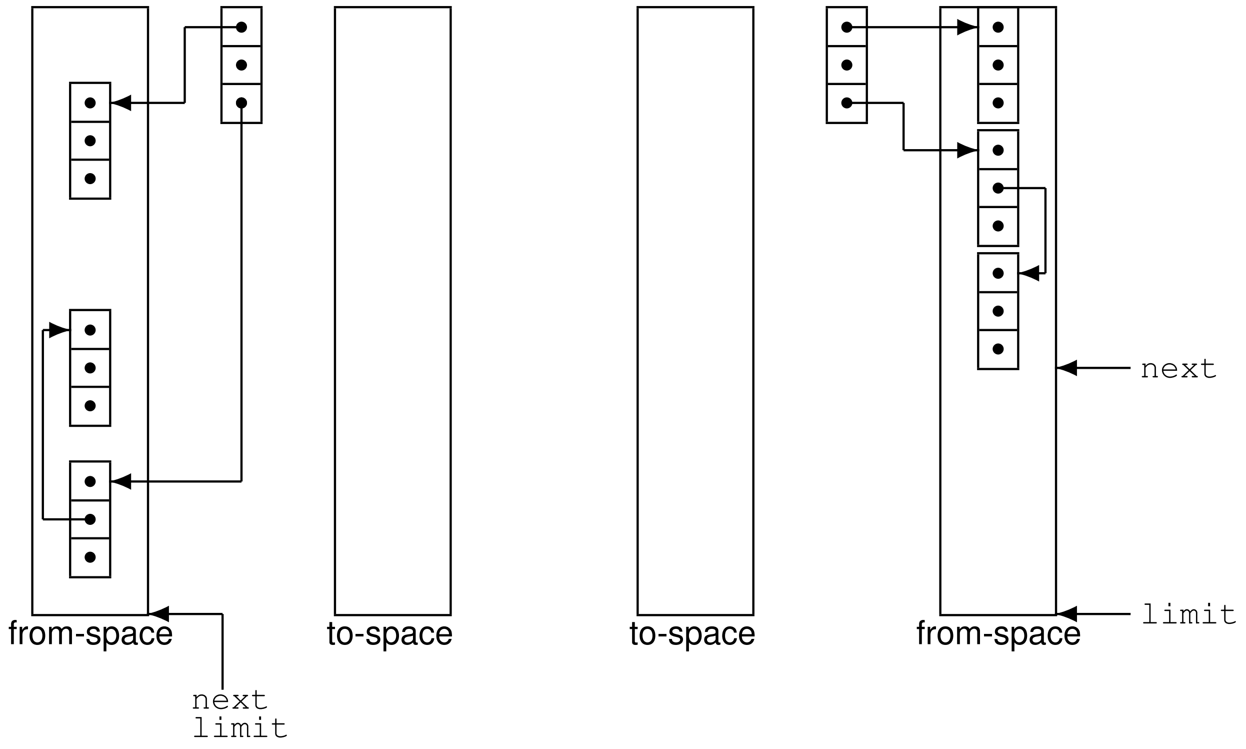
Conceptually this results in a simple high-level algorithm

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

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

The stop-and-copy algorithm internals are much more complicated. Intuitively, it *forwards* each record on the heap in a breadth-first manner (starting from the stack).

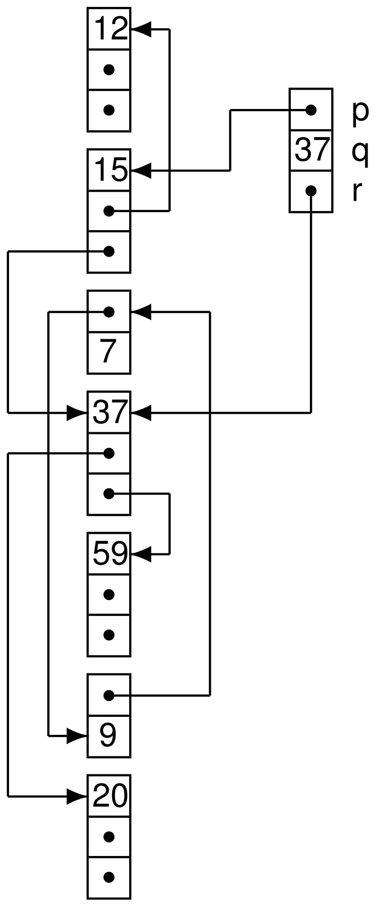
Pseudo code for stop-and-copy

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

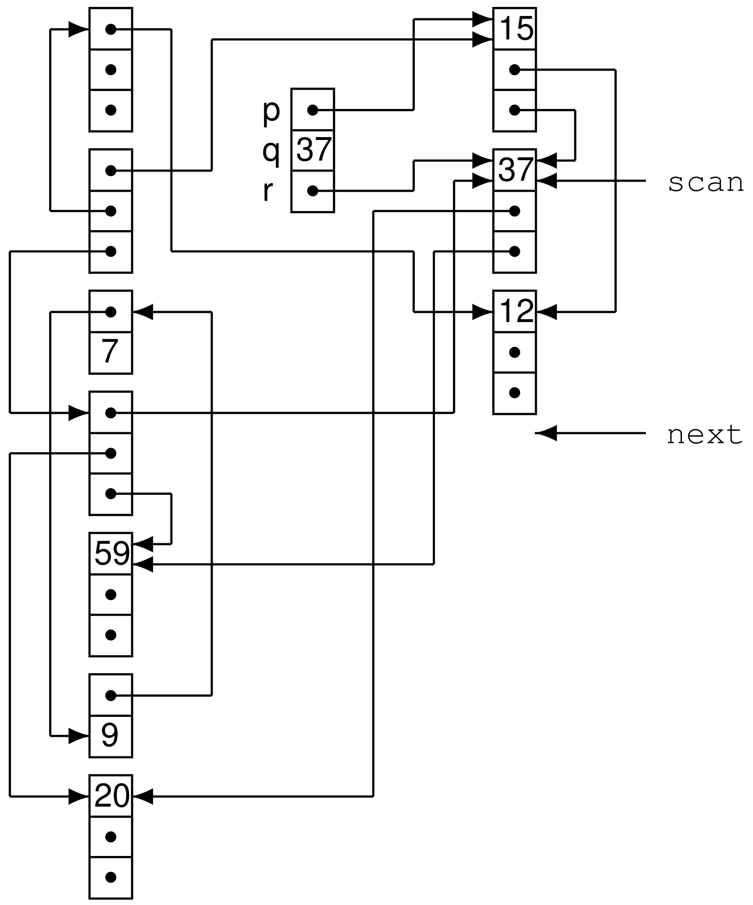
```
function Forward($p$)
    if $p \in$ from-space then
        if $p.f_1 \in$ to-space then
            return $p.f_1$
        else
            for $i$:=1 to $|p|$ do
                next.$f_i$ := $p.f_i$
            $p.f_1$ := next
            next := next + sizeof(record $p$)
            return $p.f_1$
    else return $p$
```

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

$$c_3 R$$

A realistic value

$$10R$$

The cost per reclaimed word

$$\frac{c_3 R}{\frac{H}{2} - R}$$

- This has no lower bound as H grows;
- If $H = 4R$ then the cost is $c_3 \approx 10$.

Stop-and-Copy

Advantages

- Allows fast allocation (no `freelist`);
- Avoids fragmentation;
- Collects in time proportional to $\boldsymbol{R}$; and
- Avoids stack and pointer reversal.

Disadvantage

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

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.

Practical Considerations

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

This can lead to better memory management, but garbage collection is still expensive: ≈ 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: $\boldsymbol{G_0, G_1, G_2, \ldots}$;
- All records in $\boldsymbol{G_i}$ are younger than records in $\boldsymbol{G_{i+1}}$;
- Collect $\boldsymbol{G_0}$ often, $\boldsymbol{G_1}$ less often, and so on; and
- Promote a record from $\boldsymbol{G_i}$ to $\boldsymbol{G_{i+1}}$ when it survives several collections.

Generational Collection

How to collect the G_0 generation

- It might be very expensive to find those pointers;
- Fortunately, they are rare; so
- We can try to remember them.

Ways to remember

- Maintain a list of all updated records (use marks to make this a set); or
- Mark pages of memory that contain updated records (in hardware or software).

Other Optimizations

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.

Two players access the heap

- The *mutator*: creates records and moves pointers around; and
- The *collector*: tries to collect garbage.

Some invariants are clearly required to make this work. The mutator will suffer some slowdown to maintain these invariants.