allocation and deallocation of memory is implicit.
Lisp, Scheme, ML, Erlang, Haskell, Prolog, Mercury – Both.

Module-3 – Implicit and explicit deallocation (programmer's choice).
Ada – Implicit or explicit deallocation (implementation defined).

Eiffel, Java – Allocation is explicit, but deallocation is implicit. Dynamic
and when it is safe to free it.

Pascal, C, ++, Module-2 – Allocation and deallocation are explicit.

Routines for allocation and deallocation of dynamic memory.
The runtime system linked in with the generated code should contain

Dynamic Memory Management
three sections (one for each allocation type).

The compiler and runtime system divide the available address space into

implicitly (e.g., cons).

allocated either explicitly by the programmer (new, malloc, etc.) or

3. *Heap allocation.* Used for data structures which are dynamically

and local variables.

2. *Stack allocation.* Used for activation records (procedure call chains

3. *Static allocation.* Space for global variables is allocated at compile

memory.

In a language such as C or Pascal, there are three ways to allocate

Memory Management
The heap grows by a suitable amount of memory when the program makes a request for more dynamic memory (e.g., by calling malloc).

The stack grows and shrinks during execution, according to the depth of the call chain. Deep recursion often leads to stack overflow. A large number of parameters or locals can also result in running out of stack space.

The static section is generated by the compiler and cannot be extended at run time.

Memory Management (cont.)
Memory Organization

- Heap
- Stack
- (Global variables)
- (Initialized Static Data)
- (Strings, reals, ...)
- (Constant Static Data)
- Program Code
Heap fragmentation

Memory leaks

```c
NEW(p) { g = p; DISPOSE(p); use(q); }
```

Debugging references

Problems:

Ada's new/uncHECK-deallocate (some implementations)

c malloc/free, 
c++'s new/delete,
Pascal's new/dispOSE, Module-2's ALLOCATE/DEALLOCATE

Explicit Deallocate

Fail to return a large enough block.

Fragmentation - Compact the heap as part of GC, or only when the GC

Concurrent - Run the GC as a separate thread/process.

Assessment or new.

Real-Time/Incremental - Perform a partial GC for each pointer.

Whenever we run out of space.

Disruptive - Stop execution of the program to perform GC whenever

When to deallocate data?

Explicit: Allocate, Java; Different object allocation, frequent changes to old

Tisp, Prolog - Equal-sized cells; not many changes to old cells

Implicit: Deallocation
GC is a long-studied topic (many algorithms already developed).

Higher-level.

GC frees programmers from the tedious and error-prone task of memory management, thus making a programming language with built-in GC.

GC usually requires compiler support.

GC is performed by the runtime system, not by the compiler; however,

new records. This process is called garbage collection.

Memory occupied by garbage should be reclaimed for use in allocating

from program variables are garbage.

Heap-allocated records that are not reachable by any chain of pointers
A heap to be garbage collected.

```ml
let typelist = {link: list, key: int}
let typetree = {key: int, left: tree, right: tree}

function makeTree() =
  function showTree(t: tree) =
    in
      let var x := list{link=nil, key=7}
      let var y := list{link=x, key=9}
      in
        x.link := y
    end;
    let var p := makeTree()
    let var r := p.right
    let var q := r.key
    in
      garbage-collect here
      showTree(r)
    end
  end
end

end
```

Figure 13.1.
collectors which registers/variables contain roots.

Hence, the compiler must communicate to the garbage

objects are reachable if there is a path of edges that leads to them starting

3. Local variables & formal parameters on the stack.

2. Registers;

1. Global variables;

The roots of this graph can be in:

Program variables and dynamically allocated objects form a directed graph.

Finding the Object Graph
ALGORITHM 13.2. Depth-first search

\begin{verbatim}
DFS(x)
if x is a pointer into the heap
    mark x
    if record x is not marked
        if x is a pointer into the heap
            DFS(x)

We can mark all reachable nodes using e.g. depth-first search.
\end{verbatim}
to invoke another garbage collection.

When this is exhausted, it is a good time are allocated from the freelist. After garbage collection, the program resumes execution. New records

After garbage collection, the program resumes execution. New records

a freelist (a linked list). The sweep phase also unmarks all records.

which scans the heap from beginning to end collecting all garbage cell in

Any node not marked is garbage and is reclaimed by a sweep phase.

\[
\begin{align*}
(d \text{ (size of record)} + d & \rightarrow d \\
    d & \rightarrow \text{freeList} \\
    \text{freeList} & \rightarrow f_1 \cdot d \\
    \text{if } d \text{ marked } & \text{ unmark } d \\
    \text{else let } f_1 \text{ be the first field in } d & \\
    \text{if record is marked } & \\
    \text{while last address in heap } > d & \\
    \text{first address in heap } \rightarrow d \\
\end{align*}
\]

DFS(a) for each root

Mark phase:

Sweep phase:

Mark-and-sweep Garbage Collection (code)
Figure 13.4

(b) Sweep

(a) Marked

Mark-and-Sweep Garbage Collection (Example)
allocated word will be approximately \( c_1 + 2c_2 \).

Usually, if \( H/R < H/2H \), the collector increases. Then the cost per

\[ \frac{H - H}{H^2 + c_2 R} \]

Thus, the amortized cost of GC (in instructions per allocated word) is:

\[ H \]

The "good" that GC does is to recover \( H \) words of usable memory.

Sweep phase: takes time proportional to the size of the heap, \( H \).

Marking phase: takes time proportional to the amount of reachable data.

Cost of Mark-and-Sweep Garbage Collection
freeList[i] stores all reclaimed records of size i.

Reclamation space without searching

Use of an array of freeLists allows variable-size allocation from the

records as auxiliary area (to store elements of the stack).

Pointer reversal allows use of the fields of the already processed heap

considerably smaller auxiliary area rather than activation

Use of an explicit stack allows the marking phase to require a

Turning of Mark-and-Sweep Garbage Collection
Algorithm 13.5. Depth-first search using an explicit stack.

\textbf{Function} \text{DFS}(x)

\begin{align*}
&\text{if } x \text{ is a pointer and record } x \text{ is not marked} \\
&\text{mark } x \\
&\text{if } x \text{ is a pointer and record } x \text{ is not marked} \\
&\text{foreach field } f_i \text{ of record } x \\
&\text{stack } [1] \rightarrow x \\
&\text{while } 0 < i \\
&\text{stack } x \rightarrow [1] \\
&\text{stack } i \rightarrow 1 \\
&\text{if } x \text{ is a pointer and record } x \text{ is not marked} \\
&\text{DFS}(x)
\end{align*}
function DFS(x)
    if x is a pointer and record x is not marked
        if x is a pointer and record x is not marked
            if # of fields in record x > 0
                x \mapsto \text{nil}
            end
            x \mapsto x
            x \mapsto i
            i \mapsto \text{nil}
        end
        if x is a pointer and record x is not marked
            x \mapsto \text{nil}
        end
        while true
            if x is a pointer and record x is not marked
                done \mapsto done
                if x is a pointer and record x is not marked
                    if x is a pointer and record x is not marked
                        x \mapsto \text{nil}
                    end
                end
                x \mapsto i
            end
        end
    end
end

\text{ALGORITHM 13.6. Depth-first search using pointer reversal.}
autonomic storage management in programming language environments.

Because of the disadvantages, reference counting is rarely used for

2. Keeping track of reference counts is very expensive.

1. Cycles of unreachable data cannot be detected and reclaimed.

Although reference counting seems simple and attractive, it also has

several problems:

how many pointers point to each record.

Identifying unreachable records can be done directly by the compiler

Reference Counting
With the reachable data, after a copying GC, the to-space is compact: garbage is not interspersed.

The roles of the two semi-spaces are swapped. The new part (to-space) roots are made to point to the to-space and copies the reachable data from the old part of the heap (from-space) to the allocated space is split in two semi-spaces and the garbage collector.

Copying Garbage Collector
Copyring collection.

(a) Before collection

Limit

next

next

space

to

roots

space

from

space

from

Limit

next

next

roots

space

to

Copyring Garbage Collection
Given a pointer that points from-space, make point to-to-space:

1. If \( d \) points to a from-space record that has already been copied, then it is a special forwarding pointer that indicates where the copy is (so \( d \)'s field is not needed anymore).
2. If \( d \) points to a from-space record that has not yet been copied, then it is copied to location next and the forwarding pointer is installed into \( f \). If \( f \)'s \( d \) is not a pointer at all, or points outside from-space, then forwarding does nothing.

3. If \( d \) is not a pointer at all, or points outside from-space, then
Algorithm 13.8: Forwarding a pointer

\[
\text{function } \text{Forward}(p) \text{ }
\]

\[
\text{if } p \text{ points to from-space}
\]

\[
\text{then if } p \text{ points to to-space}
\]

\[
\text{return } p \text{ from-space}
\]

\[
\text{else return } p \text{ to-space}
\]

\[
\text{else for each field } f_i \text{ of } p
\]

\[
\text{if } f_i \text{ points to from-space}
\]

\[
\text{then if } f_i \text{ points to to-space}
\]

\[
(d) \text{ Forward}(p)
\]
Cheney's Copying Garbage Collector

As depth-first search, however, breadth-first search does not give as good locality of reference.

The area between the beginning of the to-space and scan contain records that have been copied but whose fields have not yet been forwarded.

The area between scan and next is used as a queue of all records that have been forwarded.

The roots are first forwarded, reversed, and is very simple to implement.

The algorithm is non-recursive, requires no external area nor pointer to traverse the reachable data.

for each root \( r \) do
  while scan \( r \) do
    for each field \( f \) of record at scan \( r \) do
      if Forward(scan, f) then
        scan \( r \) <- next scan \( r \)
      else
        scan \( r \) <- Forward(f)
      end if
    end for
    while scan \( r \) do
      for each root \( r' \) do
        scan \( r' \) <- next scan \( r' \)
      end for
    end while
  end while
end for
Figure 13.10, From Modern Compiler Implementation in ML, Aho, Sethi, and Ullman.
In a realistic setting, \( \frac{\frac{H}{2}}{H} \approx 0 \), so GC cost is \( \approx \text{inst} \cdot \text{cost per allocated word} \).

As \( R \ll H \), the cost approaches zero, suggesting that there is no inherent lower bound to the cost of garbage collection.

\[
\frac{H - \frac{z}{H}}{\text{inst}} \leq R
\]

Each GC recovers \( \frac{R}{2} \) words of usable memory. Thus, the amortized cost of each collection (in instructions per allocated word) is:

\[
\text{GC takes time proportional to the amount of reachable data } R.
\]
Incremental collection.

Because most collections are of a small area, typical pause times are also short, and for many applications this is an acceptable alternative to collection a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

Incremental GC schemes improve efficiency and/or locality by garbage collection which proceeds on another processor in parallel with actual program execution. They can also be generalized into concurrent real-time guarantees. They can also provide a reduction in the disruption of garbage collection, and may even provide techniques that allow memory reclamation to proceed while applications are running. Incremental GC techniques allow memory reclamation to proceed.
number of GCs, they are moved into an older generation (tenured).

Once objects have survived a
areas more frequently than the older ones. Once objects have survived a
objects into multiple areas according to their age, and collecting younger

Generational collection avoids this repeated copying by segregating

Long-lived objects are saved repeatedly by a simple copying collector.

even more quickly (within tens of Kbytes of allocation).

instructions (or before a Mbyte has been allocated); the majority dies

Usually between 80–98% of all objects die within a few million

many collections will probably survive for many collections more

Newly created objects tend to die soon but objects that are reachable after
Fortunately, pointers from old to new objects are usually rare.

Remembered lists/sets, card/page marking using dirty bits.

Updates to old objects cross the write barrier. This is done by
has to ensure that all store instructions check whether (pointer)
these inter-generational references must be remembered: the compiler

These inter-generational objects in the generation being collected.
Include any pointer from objects in old generations that points into
In generational GC, roots are not just program variables; they also
 generation(s) without having to examine the older one(s).
For generational GC to work, we must be able to collect younger

**Generational Garbage Collection**
Generational collection.

(a) Before collection.

(b) After collection.
excessive than non-Generational GC.

Maintaining the remembered set also takes time; if the program updates

collections are postponed as long as possible.

Performing a major collection can be more expensive, typically such

\[
\frac{N - R}{c^3} \frac{10}{R}
\]

cost per word removed in each minor collection is:

With a copying collector, in this generation, so the amontized

10% live data.

In practice, it is common for the youngest generation to be less than

Cost of Generational Garbage Collection
generational techniques are often used as an acceptable substitute for

by users. Acceptable interactive use: most pauses are so brief that are not noticed.

For stop-and-collect garbage collection, generational GC has the

few ones that survive does not cost much.

In practice, generational GC performs quite well: the majority of objects

Characteristics of Generational GC
Incremental Garbage Collection

A concurrent GC algorithm is one which operates between or during any
instructions executed by the mutator.

When the mutator requests,
An incremental GC algorithm is one in which the collector operates only

parcels of smaller and (usually) bounded size.
Efficiently, the GC work is spread out into more uniformly distributed
reachable data.

The collector tries to collect the garbage, meanwhile, the compiled
Interleaving GC work with program execution.

Incremental (and concurrent) techniques diminish long interruptions by

Incremental Garbage Collection

♦
there are no grey objects, then all the white objects are garbage.

Starting with all objects white, objects pointed by roots are greyed. When

Cheney's algorithm these objects have been scanned).

Black objects have been marked, and their children are also marked (e.g. in

between scan and next).

Grey objects have been visited (marked or copied), but their children have

white objects are not yet visited by the depth-first or breadth-first search.

Classes of objects:

Incremental GC using Tricolor Marking

For each field $d$ of

select a grey record $d$

while there are any grey objects

if record $d$ is white

color record $d$ grey

else color record $d$ black

while there are any grey objects
and makes all children of these objects grey).

- non-black object, a page fault handler colors every object on the page black.

mutator fetches a pointer from any virtual memory page containing any

Read-barrier algorithms check all fetch instructions (e.g., whenever the

Write-barrier algorithms check every store by the mutator for

preservation of LCF2 (e.g., whenever the mutator stores a white pointer a into

- a black object q, if colors q grey).

1. No black object points to a white object.

2. Every grey object is on the collector's scheduling stack or queue.

Preserving Tricolor Invariants
newly allocated records.

At this point, no more than half the to-space will be occupied by

scan will catch up with next before next reaches half-way through the

If the heap is divided into two semi-spaces of size 2, and \( \frac{2}{H} \), then

If a pointer retired by the mutator points to from-space, the pointer is

forwarded immediately. Thus, the mutator always has pointers only to the

a pointer fetched by the mutator points to from-space, the pointer is

allocated at the end of the to-space by decrementing \( T \)\text{max}.

Process, advancing scan towards next by at least one word. New records are

Each time the mutator allocates new records, a few pointers of scan are

are swapped, and then all the roots are forwarded.

Garbage collection starts with a flip: the roles of the from-space and to-space

Based on Cheney's copying algorithm, employs a read-barrier.

Baker's Incremental Algorithm
In general, because incremental GC requires extra coordination between the mutator and the collector and higher conservatism, it is more expensive than blocking GC (where all the objects are reclaimed every time the collector is run).

The fault is processed.

Any implementation of write or read barrier must synchronize with the synchronization which is implicit in page collector. Software-based synchronization is expensive; in hardware, one can take advantage of the synchronization which is implicit in page fault: the OS ensures that no objects can access a faulting page before

**Performance of Incremental GC**
Instructions to implement the read or the write barrier.

For some versions of incremental collectors, the compiler must also generate...

describing the layout of data records on the heap (i.e., determine the...

describing locations of roots and those of them that are pointers;

Generating code which allocates records (and initializes GC);

The compiler for a garbage-collected language interacts with the garbage...

Interaction with the Compiler
scans downward, frame by frame.

To mark/forward all the roots, GC starts at the top of the stack and
a function call, the pointer map identities its register or frame location.
keyed by return addresses: for each pointer that is live immediately after
keeps by return addresses: for each pointer that is live immediately after
best

The pointer map (set of live temporaries that contain pointers) is best

The function which initiates the GC.

...the semantic analysis phase of the compiler. It is passed to the
...This descriptor is generated by the static type information calculated by
...a special type or class descriptor record.

identifying heap objects is to have the first word of each object point to
In statically typed or object-oriented languages, the simplest way of

Descriptors Data Layouts
leaks; several techniques for making these situations unlikely exist.

Conservative GC might thus occasionally suffer from disastrous space

some garbage objects might not be reclaimed.

Any bit-pattern pointing into the allocated heap is assumed to be a

record-fields contain variables, so the collector must "guess".

The compiler does not inform the collector which variables and