Memory Management
Storage

- Allocation of space for the source program data (to which variables refer)
  - Static (compile-time or load time)
  - Stack (runtime) – aka user/runtime/system stack
  - Heap (runtime)

- Storage binding for program variables - binds the address attribute of a variable to physical storage

- 3 categories of **primitive variables** (given different lifetimes)
  - Globals → static allocation
  - Locals → runtime stack allocation
  - Dynamic or heap variables (aka heap dynamic) → heap allocation
Values that a Program Manipulates Directly

- Global variables (lifetime=program) → static allocation
- Local variables (lifetime=function) → runtime stack allocation
- Heap dynamic (lifetime=programmer controlled) → heap allocation

Python3 globals
```python
gvar = 7
def foo():
    global gvar
    gvar += 1
```

Java globals/statics
```java
class C{
    final static int svar = 7;
    ...
}
```
Values that a Program Manipulates Directly

- Global variables (lifetime=program) $\rightarrow$ static allocation
- Local variables (lifetime=function) $\rightarrow$ runtime stack allocation
- Heap dynamic (lifetime=programmer controlled)) $\rightarrow$ heap allocation

Python3 globals
```python
gvar = 7
def foo():
    global gvar
    gvar += 1
```

Java globals/statics
```java
class C{
    final static int svar = 7;
    ...
}
```

Java locals
```java
void foo(){
    int lvar=7;
    ...
}
```

Python locals
```python
def foo():
    int lvar=7
    ...
```

Note here that variable values may be put into processor registers when manipulated by assembly/binary instructions (operands) ...
Values that a Program Manipulates Directly

- **Global variables** (lifetime=program) → static allocation
- **Local variables** (lifetime=function) → runtime stack allocation
- **Heap dynamic** (lifetime=programmer controlled) → heap allocation

**Python3 globals**
```python
def foo():
global gvar
gvar += 1
```

**Java globals/statics**
```java
class C{
    final static int svar = 7;
    ...
}
```

**Python3 heap dynamic**
```python
class C: ...
    def __init__(self,val):
        self.inst_field = val
x = C(100) //new C:
    //init w/arg=100
```

**Java locals**
```java
void foo(){
    int lvar=7;
    ...
}
```

**Python locals**
```python
def foo():
    int lvar=7
    ...
}
```

**Java heap dynamic**
```java
class C {
    C(int val) {}
} ...
C myC = new C(100);
```

Note here that variable values may be put into processor registers when manipulated by assembly/binary instructions (operands) ...

Note here that variable is global or local and holds a reference/address

Address held is the object in heap
An OS Process

Executable & Linkable Format (ELF): Linux/GNU

• An executable file that has been loaded into memory
  - The OS has been told that the file is ready (**exec** command)
  - OS schedules it for execution (to get a turn using the CPU)

• Since we are using virtual memory (paging physical memory pages between virtual memory and disk)
  - A process has its own **address space**
    - Provides isolation of processes (a process cannot access an address in another process)

Process Memory (virtual address space)

- high memory

OS kernel virtual memory

User program
-- in our case this is the language virtual machine which then loads and executes a user program as part of its address space

Unused/Not Accessible

Address 0xffffffff

Address 0x0

low memory
An OS Process
Executable & Linkable Format (ELF): Linux/GNU

- An executable file that has been loaded into memory
- Has its own process address space in virtual memory
  - Broken up into segments
- Other names for runtime stack (holds the args passed in, return address, and local variables of the function being called):
  - Stack
  - User stack
  - Machine stack
  - Call stack

<table>
<thead>
<tr>
<th>Process Memory (virtual address space)</th>
</tr>
</thead>
<tbody>
<tr>
<td>high memory</td>
</tr>
<tr>
<td>OS kernel virtual memory</td>
</tr>
<tr>
<td>User/Runtime stack</td>
</tr>
<tr>
<td>To support function execution</td>
</tr>
<tr>
<td>and local variable storage</td>
</tr>
<tr>
<td>Shared library region</td>
</tr>
<tr>
<td>Runtime heap</td>
</tr>
<tr>
<td>For dynamic (explicit) allocation</td>
</tr>
<tr>
<td>of dynamic variables</td>
</tr>
<tr>
<td>Read/write segment</td>
</tr>
<tr>
<td>For globals (&amp; static locals)</td>
</tr>
<tr>
<td>Read-only segment</td>
</tr>
<tr>
<td>For program code and constant</td>
</tr>
<tr>
<td>global variables</td>
</tr>
<tr>
<td>Unused/Not Accessible</td>
</tr>
<tr>
<td>low memory</td>
</tr>
</tbody>
</table>

Address 0xffffffff
Address 0x0
Address 0x0xffffffff
Address 0x0
Variable Storage and Lifetime

- Time a variable is bound to a particular memory location
- 3 categories of primitive variables (given different lifetimes)
  - **Globals (static storage)**
    - Variables declared outside of any function or class (outermost scope)
    - Scope: accessible to all statements in all functions in the file
    - Lifetime: from start of program (loading) to end (unloading)
    - Good practice: use sparingly, make constant as often as possible
    - Stored in **read-only or read-write** segments of the process virtual memory space – allocated/fixed before program starts
      - Read-only segment holds translated/native code as well if any
        - Note that this is the native code of the **runtime system process**
Variable Storage and Lifetime

- Time a variable is bound to a particular memory location
- 3 categories of primitive variables (given different lifetimes)
  - **Globals (static storage)**
    - Variables declared outside of any function or class (outermost scope)
    - Scope: accessible to all statements in all functions in the file
    - Lifetime: from start of program (loading) to end (unloading)
    - Good practice: use sparingly, make constant as often as possible
    - Stored in **read-only** or **read-write** segments of the process virtual memory space – allocated/fixed before program starts
      - Read-only segment holds translated/native code as well if any
        - Note that this is the native code of the **runtime system process**
          - When a runtime system dynamically compiles coded from a high-level language program (java/javascript/.Net/python), it puts it **in the heap**
            - i.e. each method for method-level compilation (java) and in a code cache for trace compilation (javascript)
An OS Process

- An executable file that has been loaded into memory
- Has its own process address space in virtual memory
  - Broken up into segments

- Runtime system code in read-only segment
  - The high-level program code, if compiled (methods, code cache) goes in the Runtime heap

<table>
<thead>
<tr>
<th>Process Memory (virtual address space)</th>
</tr>
</thead>
<tbody>
<tr>
<td>high memory</td>
</tr>
<tr>
<td>OS kernel virtual memory</td>
</tr>
<tr>
<td>User/Runtime stack</td>
</tr>
<tr>
<td>To support function execution and local variable storage</td>
</tr>
<tr>
<td>Shared library region</td>
</tr>
<tr>
<td>High-level language compiled code</td>
</tr>
<tr>
<td>(if any)</td>
</tr>
<tr>
<td>Runtime heap</td>
</tr>
<tr>
<td>For dynamic (explicit) allocation of dynamic variables</td>
</tr>
<tr>
<td>Read/write segment</td>
</tr>
<tr>
<td>For globals (&amp; static locals)</td>
</tr>
<tr>
<td>Read-only segment</td>
</tr>
<tr>
<td>For program code and constant global variables</td>
</tr>
<tr>
<td>Unused/Not Accessible</td>
</tr>
<tr>
<td>low memory</td>
</tr>
</tbody>
</table>

Executable & Linkable Format (ELF): Linux/GNU

Address 0xffffffff

Address 0x0
Variable Storage and Lifetime

- Time a variable is bound to a particular memory location
- 3 categories of primitive variables (given different lifetimes)
  - Globals (static storage)
  - Locals (stack storage)
    - **Parameters and variables** declared within a function
    - Scope: accessible to all statements in the function they are defined
    - Lifetime: from start to end of the function invocation
    - Stored in User/Runtime stack in process virtual memory space
      - Allocated/deallocated with function invocations and returns
      - In a runtime system for high-level languages, this is shared between the runtime process and the high-level language program
Running Interpreter (VM) Memory: Javascript V8

Process Runtime Heap:

For the runtime system: global statics/symbol table, internal (code/data/obj) representations and maps, GC book keeping data structures, ...

For the program: bytecode files read in, compiled code bodies/code cache, objects allocated during execution
Variable Storage and Lifetime

- Time a variable is bound to a particular memory location
- 3 categories of primitive variables (given different lifetimes)
  - Globals (static storage)
  - Locals (stack storage)
  - Dynamic variables, aka pointer variables (heap storage)
    - Pointer variables that point to variables that are allocated explicitly
    - Scope: global or local depending on where they are declared
    - Lifetime: from program point at which they are allocated with `new` to the one at which they are deallocated with `delete`

- Pointer variables (the address) are either globals or locals
- The data/objects they point to is stored in the heap segment of the process’ virtual memory space
  - Objects allocated by high-level language program
  - Objects/data structures needed by the runtime system process
    - statics table, method bodies, code cache, internal reps of classes
Heap Allocation and Deallocation

Explicit allocation and deletion

- New, malloc, delete, free
- Programmer controls all
  - Delete an object following the last use of it
Heap Allocation and Deallocation

- Explicit allocation and deletion
  - New, malloc, delete, free
  - Programmer controls all
    - Delete an object following the last use of it

- Implicit
  - Programmers do nothing, its all automatic
  - Non-heap objects are implicitly allocated and deallocated
    - Local variables, deallocated with simple removal of user stack frame
    - Globals, never deallocated, cleaned up with program at end
  - Implicit deallocation of heap objects
    - Garbage collection
      - May **not** remove an object from system immediately after its last use

- Stack variables (locals and params), static variables (globals) use implicit allocation and deallocation
Failures in Explicitly Deallocated Memory

= programmer must free explicitly allocated heap memory (use delete to free new’d objects)

• Memory leaks
• Dangling pointers
• Out of memory errors
• Errors may not be repeatable (system dependent)
• Dynamic memory management in complex programs is very difficult to implement correctly
  ■ Even for simple data structures
• Multi-threading/multi-processing complicates matters

• Debugging/tracing is very difficult (requires other tools)
  ■ Purify
  ■ But these only work on a running program (particular input and set of paths taken are the only ones checked)
Garbage Collection

- Solves explicit deallocation problems through automation
- Introduces runtime processing (overhead) to do the work
- Not the solution to every problem in any language
  - However it is REQUIRED for managed languages
    - For which programs can be sandboxed to protect the host system

- But it will
  - Reduce the number of bugs and hard to find programming errors
  - Reduce program development/debugging cycle

- However, it should be an integrated part of the system
  - Not an afterthought or hack

- May even improve performance! ... How?
Garbage Collection

- Key component of modern runtime systems
  - Enabling programmer productivity and memory safety
- Runs while the program is executing
  - Allocates memory for the program (new Object();) when requested
  - Deallocates memory when unreachable by the program
    - 
      - **Reachable** = graph of “live” objects with roots in the program variables (those with reference types)
      - May occur as soon as object becomes unreachable
        - Or some time thereafter
Where GC Fits in the Runtime System

- Program loaded from disk/network
  - Statics table populated, Internal class reps/maps created
- Methods invoked
  - Compiled or interpreted
- Method execution employs
  - New (space is allocated from the process heap)
  - And pointer assignments
  - GC (allocator and collector) handles both for the program
  - GC and program execution are interleaved
    - Collector may execute using multiple threads
Terminology

- **Mutator**
  - User program - change (mutate) program data structures

- **Collector**
  - Memory manager – invoked when the program runs out of memory or objects become unreachable
    - b/c of the former: should not allocate memory!

- **Stop-the-world** collector - all *mutators stop* during GC
  - simplest form of GC (+ in wide spread use)
Terminology

- **Values (reference types) that a program can manipulate directly**
  - in CPU/processor registers (hold temporary data: native instruction operands)
  - on runtime stack (includes locals/parameters/spills/temporaries)
  - Globals (e.g., data in statics table)

- **Root set** of the computation (program)
  - References to heap data held in these locations
    - Dynamically allocated data **only** accessible via roots
      - A program should not access random locations in heap
  - “Live” objects are those reachable by the **roots** (all else is garbage) – we build a **live graph** of these objects
    - Note that there is a root set for each program point (assembly instruction) during execution
    - Garbage collector can reclaim/clean up anything **not reachable**
      - What is this called in a system **without** garbage collection?
GC Example

mutator

```java
static MyList listEle;
void foo() {
    listEle = new MyList();
    listEle.next = new MyList();
    listEle.next.next = new MyList();
    MyList localEle = listEle.next;
    listEle = null;
    Object o = new Object();
}
GC Example

```java
mutator

    static MyList listEle;
    void foo() {
        listEle = new MyList();
        listEle.next = new MyList();
        listEle.next.next = new MyList();
        MyList localEle = listEle.next;
        listEle = null;
        Object o = new Object();
    }
```

Root Set: statics, stack vars, registers

GC Cycle

1. Detection
2. Reclamation

Restart mutators
Liveness of Allocated Objects

- Determined *indirectly* or *directly*
- Indirectly
  - Most common method: *tracing*

- Regenerate the set of live nodes whenever a request by the user program for more memory fails

  - Start from each root and visit all reachable nodes (via pointers)

- Any node not visited is reclaimed
Liveness of Allocated Objects

- Determined indirectly or directly
- Directly
  - A record is associated with each node/obj in the heap and all references to that node from other heap nodes or roots

- Most common method: **reference counting**
  - Store a count of the number of pointers to this cell in the cell itself

- Must be kept up to date as the mutator alters the connectivity of the heap graph
Three Classic Garbage Collection Algorithms

- Reference counting
- Mark & Sweep
- Copying
Three Classic Garbage Collection Algorithms

- Reference counting
- Mark & Sweep
- Copying

Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object

Free List Allocation is also used by C/C++
The Free List: Internal VM/runtime data structure – linked list of free blocks
- Memory is **one big contiguous array**
  - In the virtual address space of the processor: Runtime Heap area
  - Typically word-aligned addresses

- Objects = data allocated in memory
  - With header and fields (as discussed previously)
  - We’ll assume 2 fields in all objects in the following slides
An OS Process

Executable & Linkable Format (ELF): Linux/GNU

- An executable file that has been loaded into memory

- Has its own process address space in virtual memory
  - Broken up into segments

<table>
<thead>
<tr>
<th>OS kernel virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>User/Runtime stack</td>
</tr>
<tr>
<td>To support function execution and local variable storage</td>
</tr>
<tr>
<td>Shared library region</td>
</tr>
<tr>
<td>Free list + GC'd area</td>
</tr>
<tr>
<td>Runtime heap</td>
</tr>
<tr>
<td>For dynamic (explicit) allocation of dynamic variables</td>
</tr>
<tr>
<td>Read/write segment</td>
</tr>
<tr>
<td>For globals (&amp; static locals)</td>
</tr>
<tr>
<td>Read-only segment</td>
</tr>
<tr>
<td>For program code and constant global variables</td>
</tr>
<tr>
<td>Unused/Not Accessible</td>
</tr>
</tbody>
</table>

Address 0xffffffff

Address 0x0
The Free List: Internal VM/runtime data structure – linked list of free blocks
- Memory (virtual/heap) is one big contiguous array
- Typically multiple lists (each with different sized blocks – e.g. powers of 2)
  - Linked together in a linked list (hidden next pointer + list_head)
The Free List: Internal VM/runtime data structure – linked list of free blocks
- Memory (virtual/heap) is one big contiguous array
- Typically multiple linked lists (of different sizes)
  - Allocation takes the chuck of list that is ≥ the size needed
  - Deallocation/free puts them on the front for reuse (in cache)
- When a partial block is used, the remainder gets put back on a list (acc. to size)
- When two blocks are next to each other, they can be combined
- Multiple allocations and frees can/will cause fragmentation
  - slows down allocation process: hard to find a hole big enough...
h = new LARGE_OBJECT()
Roots: a b c d e f g h i j k l

h = new LARGE_OBJECT()
d = NULL (before)
h = new LARGE_OBJECT()
d = NULL (before)
Roots: a b c d e f g h i j k l

h = new LARGE_OBJECT()
d = NULL (after)
Questions?
Three Classic Garbage Collection Algorithms

- **Reference counting**
- **Mark & Sweep**
- **Copying**

Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object
The Free List: Internal VM/runtime data structure – linked list of free blocks
- Memory (virtual/heap) is one big contiguous array
- Typically multiple lists (each with different sized blocks – e.g. powers of 2)

  - Linked together in a linked list (hidden next pointer + list_head)
**The Free List:** Internal VM/runtime data structure – linked list of free blocks
- Memory (virtual/heap) is one big contiguous array
- Typically multiple linked lists (of different sizes)
  - Allocation takes the chunk of list that is ≥ the size needed
  - Deallocation/free puts them on the front for reuse (in cache)
- When a partial block is used, the remainder gets put back on a list (acc. to size)
- When two blocks are next to each other, they can be combined
- Multiple allocations and frees can/will cause **fragmentation**
  - slows down allocation process: hard to find a hole big enough...
Reference Counting GC Algorithm

- Each object has an additional atomic field in header
  - **Reference count**
    - Holds number of pointers to that cell from roots or other objects
- All cells placed in free list initially with count of 0
- **Free_list** points to the head of the free list

- Each time a **pointer is set** to refer to this cell, the count is incremented
- Each time a reference is removed, count is decremented
  - If the count goes to 0
    - There is no way for the program to access this cell
  - The cell is returned to the free list
Reference Counting GC Algorithm

- When a new cell is allocated
  - Reference count is set to 1
  - Removed from free list

  - Assume, for now, that all cells are the same size and each has 2 fields left and right which are references

Allocate() {
  newcell = free_list
  free_list = free_list->next
  return newcell
}

New() {
  if (free_list) == NULL
    abort("Out of Memory")
  newcell = allocate()
  newcell->RC = 1
  return newcell
}

R = New():

Free(N) {
  N->next = free_list
  free_list = N
}

Delete(T) {
  T->RC--
  if (T->RC == 0) {
    for U in Children(T)
      Delete (*U)
    Free(T)
  }
}

Update(R->left,S) {
  // we assume R,S are pointers
  S->RC++
  Delete(*R)
  *R = S
}

Update(R,S) {
  // we assume R,S are pointers
  S->RC++
  Delete(*R)
  *R = S
}

R->left = S

R = New():
Reference Counting GC Algorithm: Update

- Assume, for now, that all cells are the same size and each has 2 fields left and right which are references

```c
Free(N) {
    N->next = free_list
    free_list = N
}
```

```c
Delete(T) {
    T->RC--
    if (T->RC == 0) {
        for U in Children(T)
            Delete(*U)
        Free(T)
    }
}
```

```c
Update(R,S) {
    // we assume R,S are pointers and nulls are handled correctly
    S->RC++ (if !NULL)
    Delete(*R)
    *R = S
}
```

**R->right = NULL**

Before Update(R->right,NULL)

Before if in Delete(*(R->right))
Reference Counting GC Algorithm: Update

Assume, for now, that all cells are the same size and each has 2 fields left and right which are references.

Free(N) {
   N->next = free_list
   free_list = N
}

Delete(T) {
   T->RC--
   if (T->RC == 0) {
      for U in Children(T)
         Delete(*U)
      Free(T)
   }
}

Update(R,S) {
   // we assume R,S are pointers and nulls are handled correctly
   S->RC++
   Delete(*R)
   *R = S
}

Delete(S->left) --> After: Update(R->right,NULL)
Reference Counting GC

- Strengths
  - Memory management overheads are distributed throughout the computation
    - Management of active and garbage cells is interleaved with execution
    - Incremental
    - Smoother response time
  - Locality of reference
    - Things related are accessed together (for mem.hierarchy perf.)
    - No worse than program itself
  - Short-lived cells can be reused as soon as they are reclaimed
    - We don’t have to wait until memory is exhausted to free cells
    - Immediate reuse generates fewer page faults for virtual memory
    - Update in place is possible
Reference Counting GC

- Weaknesses
  - High processing cost for each pointer update
    - When a pointer is overwritten the reference count for both the old and new target cells must be adjusted
    - May cause poor memory performance
    - Hence, it is not used much in real systems
  - Fragile
    - Make sure to get all increments/decrements right
    - Increment for each call in which a pointer is passed as a parameter
    - Hard to maintain
  - Extra space in each cell to store count
    - Size = the number of pointers in the heap = sizeof(int)
    - Alternative: smaller size + overflow handling
  - Free list: fragments memory
Reference Counting GC

- Weaknesses
  - Cyclic data structures can’t be reclaimed
    - Doubly linked lists
    - Solution: reference counting + something else (tracing)

This cycle is neither reachable nor reclaimable

Delete(R->right)
Three Classic Garbage Collection Algorithms

- Reference counting
- **Mark & Sweep**
- Copying

Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object
Mark & Sweep GC Algorithm

- Tracing collector
  - Mark-sweep, Mark-scan
  - Use reachability (indirection) to find live objects

- Objects are **not reclaimed** immediately when they become garbage
  - Remain unreachable and undetected until storage is exhausted
Mark & Sweep GC Algorithm

- Tracing collector
  - Mark-sweep, Mark-scan
  - Use reachability (indirection) to find live objects
- Objects are **not reclaimed** immediately when they become garbage
  - Remain unreachable and undetected until storage is exhausted
- When reclamation happens the program is paused
  - **Sweep** all currently unused cells back into the free_list
  - GC performs a global traversal of all live objects to determine which cells are reachable (**live or active**)
    - Trace, starting from roots, marking them as reachable
    - Free all unmarked cells
Mark & Sweep GC Algorithm

- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```
New() {
    if free_list->is_empty()
        mark_sweep
    newcell = allocate()
    return newcell
}

mark(N) {
    if N->mark_bit == 0
        N->mark_bit = 1
    for M in Children(N)
        mark(M)
}

sweep() {
    N = heap_start
    while (N < heap_end) {
        if N->mark_bit == 0
            free(N)
        else N->mark_bit = 0
        N+=sizeof(N)
    }
}
```

The heap graph of objects
Mark & Sweep GC Algorithm

- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```
New() {
    if free_list->isEmpty() {  
        mark_sweep
        newcell = allocate()
        return newcell
    }
}

mark(N) {
    if N->mark_bit == 0 {
        N->mark_bit = 1
        for M in Children(N) {
            mark(M)
        }
    }
}

mark_sweep() {
    for R in Roots {
        mark(R)
        sweep()
    }
}

sweep() {
    N = heap_start
    while (N < heap_end) {
        if N->mark_bit == 0 {
            free(N)
        } else {
            N->mark_bit = 0
            N += sizeof(N)
        }
    }
}
```

The heap graph after the marking phase, all unmarked cells are garbage
Mark & Sweep GC Algorithm

- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```c
New() {
    if free_list->isEmpty()
        mark_sweep
    newcell = allocate()
    return newcell
}

mark_sweep() {
    for R in Roots
        mark (R)
    sweep()
    if free_list->isEmpty()
        abort("OutOfMemory")
}

mark(N) {
    if N->mark_bit == 0
        N->mark_bit = 1
    for M in Children(N)
        mark(M)
}

sweep() {
    N = heap_start
    while (N < heap_end) {
        if N->mark_bit == 0
            free(N)
        else N->mark_bit = 0
        N+=sizeof(N)
    }
}
```

All of the gray areas are skipped (but considered) during sweeping.

The heap graph after the marking phase, all unmarked cells are garbage.
Mark & Sweep GC Algorithm

- Each cell contains 1 bit (mark_bit) of extra information
- Cells in free_list have mark_bits set to 0
- No Update(...) routine necessary

```
New() {
    if free_list->isEmpty() {
        mark_sweep()
        newcell = allocate()
        return newcell
    }
}

mark(N) {
    if N->mark_bit == 0 {
        N->mark_bit = 1
        for M in Children(N) {
            mark(M)
        }
    }
}

mark_sweep() {
    for R in Roots {
        mark (R)
    }
    sweep()
    if free_list->isEmpty() {
        abort("OutOfMemory")
    }
}

sweep() {
    N = heap_start
    while (N < heap_end) {
        if N->mark_bit == 0 {
            free(N)
        } else {
            N->mark_bit = 0
            N += sizeof(N)
        }
    }
}
```

The heap graph after the sweeping phase, all unmarked cells are live
Mark & Sweep GC Algorithm

- **Strengths**
  - Cycles are handled quite normally
  - No overhead placed on pointer manipulations
  - Better than (incremental) reference counting

- **Weaknesses**
  - Start-stop algorithm (aka stop-the-world)
    - Computation is halted while GC happens
    - Not practical for real-time systems
  - Asymptotic complexity is proportional to the size of the heap
    - not just the live objects
      - For sweep
Mark & Sweep GC Algorithm

• Weaknesses (continued)
  ■ Free list (like for Reference Counting): **Fragments** memory
    (scatters free cells across memory)
    ▸ Loss of memory performance (caching/paging)
    ▸ Allocation is complicated (need to find a set of cells for the right size)
  ■ **Residency** - heap occupancy
    ▸ As this increases, the need for garbage collection will become more frequent
    ▸ Taking processing cycles away from the application
    ▸ Allocation and program performance degrades as residency increases
      ◆ Overhead depends on program behavior
Mark-Compact

- Mark-sweep with compaction
- Compact live data during reclamation

Advantages
- Zero fragmentation (after compaction)
- Fast allocation – Increment a pointer into free space
- Improved locality

Disadvantages
- At least two passes over entire heap required during compaction
Questions?
Three Classic Garbage Collection Algorithms

- Reference counting
- Mark & Sweep
- Copying

Free List Allocation: keep 1+ lists of free chunks that we then fill or break off pieces of to allocate an object

Bump-Pointer Allocation: increment a pointer to get the next chunk of memory for an object being allocated
Reference Counting GC Algorithm

• Each object has an additional atomic field in header
  ■ Reference count
    ▶ Holds number of pointers to that cell from roots or other objects
• All cells placed in free list initially with count of 0
• Free_list points to the head of the free list
• Each time a pointer is set (during program execution) to refer to this cell, the count is incremented
• Each time a reference is removed, count is decremented
  ■ If the count goes to 0
    ▶ There is no way for the program to access this cell
  ■ The cell is returned to the free list
Mark & Sweep GC Algorithm

- **Tracing collector**
  - Mark-sweep, Mark-scan
  - Use reachability (indirection) to find live objects

- **Objects are not reclaimed immediately when they become garbage**
  - Remain unreachable and undetected until storage is exhausted

- **When reclamation happens the program is paused**
  - **Sweep** all currently unused cells back into the free_list
  - GC performs a global traversal of all live objects to determine which cells are reachable *(live or active)*
    - Trace, starting from roots, marking them as reachable
    - Free all unmarked cells

- **We can add a Compact phase (Mark-Compact) to remove fragmentation every once in awhile**
Copying Collector

- Tracing, stop-the-world collector
  - Divide the heap into two **semispaces**
    - One with current data
    - The other with obsolete data
  - The roles of the two semispaces is continuously **flipped**
  - Collector copies live data from the old semispace
    - FromSpace
    - To the new semispace (ToSpace) when visited
    - Pointers to objects in ToSpace are updated
    - Program is restarted

- **Scavengers**
  - FromSpace is not reclaimed, just abandoned
Copying Collection

- **Advantages**
  - Fast allocation – Increment a pointer into free space
    - Bump pointer allocation
  - No fragmentation

- **Disadvantages**
  - Available heap space is **halved**
  - Large copying cost
  - **Locality** not always improved
Copying Collector

InitGC() {
  ToSpace = heap_start
  space_size = heap_size/2
  top_of_space = ToSpace+space_size
  FromSpace = top_of_space+1
  freeptr = toSpace
}

New(n) {
  if freeptr+n > top_of_space
    flip()
  if freeptr+n > top_of_space
    abort(“OutOfMemory”)
  newcell = freeptr
  freeptr = freeptr+n
  return newcell
}

flip() {
  FromSpace, ToSpace = ToSpace, FromSpace
  top_of_space = ToSpace+space_size
  freeptr = ToSpace
  for R in Roots
    R = copy(R)
}

AKA: the bump pointer
copy(P)
if P == NULL || P->is_not_object
    return P
else if !forwarded(P) {
    n = size(P)
    P' = freeptr
    freeptr = freeptr + n
    forwarding_address(P) = P'
    for (i = 0; i < child_objs; i++)
        P'[i] = copy(P[i]);
} else {
    return forwarding_address
}
copy(P)
  if P == NULL || P->is_not_object
    return P
  if !forwarded(P) {
    n = size(P)
    P' = freeptr
    freeptr = freeptr+n
    forwarding_address(P) = P'
    for (i = 0; i<child_objs; i++)
      P'[i] = copy(P[i]);
  }
  return forwarding_address
}
Copying (Semispace) Collector

root

FromSpace

ToSpace

A

B

C

D

A'

B'

C'

D'

1

2

. . .
Copying (Semispace) Collector

root

A
B
C
D

ToSpace

1
2

FromSpace

A'
B'
C'
D'
Copying (Semispace) Collector

\begin{verbatim}
copy(P)
  if P==NULL || P->is_not_object
    return P
  if !forwarded(P) {
    n = size(P)
    P' = freeptr
    freeptr = freeptr+n
    forwarding_address(P) = P'
    for (i = 0; i<child_objs; i++)
      P'[i] = copy(P[i]);
  }
  return forwarding_address
\end{verbatim}
Copy (Semispace) Collector

```c
void copy(P)
    if P == NULL || P->is_not_object
        return P;
    if !forwarded(P) {
        n = size(P);
        P' = freeptr;
        freeptr = freeptr + n;
        forwarding_address(P) = P';
        for (i = 0; i < child_objs; i++) {
            P'[i] = copy(P[i]);
        }
    }
    return forwarding_address;
```
Copying Collector

- **Strengths**
  - Have lead to its widespread adoption
  - Active data is compact (not fragmented as in mark-sweep)
    - More efficient allocation, just grab the next group of cells that fits
    - The check for space remaining is simply a pointer comparison
  - Handles variable-sized objects naturally
  - No overhead on pointer updates
  - Allocation is a simple free-space pointer increment
  - Fragmentation is eliminated
    - Compaction offers improved memory hierarchy performance of the user program
Copying Collector

- Weaknesses
  - Required address space is doubled compared with non-copying collectors
    - Primary drawback is the need to divide memory into two
    - Performance degrades as residency increases (twice as quickly as mark&sweep b/c half the space)
  - Touches every page (VM) of the heap regardless of residency of the user program
    - Unless both semispaces can be held in memory simultaneously
Other Popular GCs

• Observations with previous GCs
  - Long-lived objects are hard to deal with
  - Young objects (recently allocated) die young
    ▶ Most are young (80-90%) = weak-generational hypothesis
  - Large heaps (that can’t be held in memory) degrade perf.

• Goal: Make large heaps more efficient by concentrating effort where the greatest payoff is

• Solution: Generational GC
  - Exploit the lifetime of objects to make GC and the program’s use of the memory hierarchy more efficient
Generational GC

• Segregate objects by age into two or more heap regions
  ■ Generations
    ▸ Keep the young generation separate
  ■ Collected at different frequencies
    ▸ The younger the more often
    ▸ The oldest, possibly never

• Can be implemented as an incremental scheme or as a stop-the-world scheme
  ■ Using different algorithms on the different regions

• Ok - so how do we measure life times?
Measuring Object Lifetimes

- **Time?**
  - Machine dependent - depend on the speed of the machine
  - Alternative 1: Number of instructions executed - also dependent across instruction set architectures
  - Alternative 2: Number of bytes allocated in the heap
    - Machine dependent
    - But gives a good measure of the demands made on the memory hierarchy
    - Closely related to the frequency of collection

- Problems
  - In interactive systems, this can be dependent upon user behavior
  - Language and VM dependent
Generational GC

- Allocate from young/nursery area
  - Except perhaps for large or known-to-be-long-lived objects
- When it fills, collect only the young/nursery space
  - *Minor collection*
    - Independently, without collecting the older space(s)
Generational GC

- Promotion (during a minor collection)
  - Move object to older generation if its survives long enough
- Concentrate on youngest generation for reclamation
  - *Most should be dead, so this should be very fast & productive*
  - Make this region **small** so that its collection can be more **frequent** but with **shorter** interruption

![Diagram of generational garbage collection](image.png)
Generational GC

- Collect young gen without collecting old gen
  - The pause time to collect a younger gen. is shorter than if a collection of the heap is performed
- **Tenured garbage** - garbage in older generations
- When the old space fills, perform a full heap (major) collection: minor collection then old space collection

![Diagram of generational GC]

- **Diagram Description**
  - The diagram represents the generational garbage collection process.
  - Nodes A, B, C, and R are part of the root set.
  - The old and young generations are divided by a line.
  - Arrows indicate the flow of objects from the young to the old generation.
Generational GC

- Allocate large or known-to-be-old objects directly to old
- Minor is frequent, Major is very infrequent
- Major/minor collections can be any type
  - Mark/sweep, copying, mark/compact, hybrid
  - Promotion is copying
- Can have more than 2 generations
  - Each requiring collection of those lower/younger
Generational Collection

- Minor Collection must be **independent** of major
  - Need to remember old-to-young references
  - Usually not too many – mutations to old objects are infrequent

Root: R1, R2

Old (mature space)  Young (nursery)
Generational Collection

- Minor Collection must be \textbf{independent} of major
  - Need to remember old-to-young references
  - Usually not too many – mutations to old objects are infrequent

- **Write Barrier**
  - Check pointer stores
  - Remember source object
  - Source object is root for minor GC
Generational Collection

- Minor Collection must be independent of major
  - Need to remember old-to-young references
  - Usually not too many – mutations to old objects are infrequent

![Diagram of Generational Collection]

- Write Barrier
  - Check pointer stores
  - Remember source object
  - Source object is root for minor GC
Generational GC

- What about young-to-old?
Generational GC

• What about young-to-old?
  ■ We don’t need to worry about them if we always collect the young each time we collect the old (major collection)
    ▸ Major: chase all roots to both young, old; finding all live objects
Other Things to Consider...

- Pig in the snake problem
  - Relatively longlived objects (together make up a large portion of the heap) become garbage all at once
  - Will be copied repeatedly & until space for it is found
  - Increases traversal cost at every generation
  - **Favors fast advancement of large object clusters**

- Questions
  - More generations?
  - How big should they be?
  - How can we make things more efficient?
Advanced GC Topics

- Parallel collection
- Concurrent collection
The Principle of Locality

- A good GC should not only reclaim memory but improve the locality of the system on the whole
  - Principle of locality - programs access a relatively small portion of their address space at any particular time
    - Spacial locality - if an object is referenced, the objects near to it (in space) are likely to also be referenced soon
    - Temporal locality - if an object is referenced, it is likely to be referenced again soon
  - GC should ensure that locality is exploited to improve performance wherever possible

- Memory hierarchy was developed to exploit the natural principle of locality in programs
  - Different levels of memory each with different speeds/sizes/cost
  - Registers, cache, memory, virtual memory
Parallel/Concurrent Garbage Collection

- **Parallel** – multi-threaded collection (scalability on SMP/multi-core)
  - GC still stop-the-world

- **Concurrent** – unlike stop-the-world (STW), background collection (short pauses through resource over-provisioning)
Advanced GC Topics

- Parallel collection
- Concurrent collection

Sun HotSpot (OpenJDK) GC

- Generational mark-sweep/compact
- Eden: where objects are allocated via bump pointer
  - When full, live objects copied to To space
- To: half of nursery; From: half of nursery
  - Flip spaces here 2-3 times (parameter setting)
- Mature space: Mark-sweep with region-based compaction

Advanced GC

- Partner with operating system: Mapping Collector
Immix

- Mark-region, parallel stop the world collection
  - Sweep to region strategy – To get bump pointer allocation
  - Full heap (Immix) or generational’s mature space (GenIX)

- Regions: blocks and lines
  - 32K Block size, 128B Line size; objects can span lines, not blocks; estimated marking
  - Large objects are >256B, give own block, exact marking

- Opportunistic defragmentation
  - Single pass: Leaves forwarding pointers when moved
  - Stops if it runs out of space to continue
  - Leaves pinned objects in place
Uses page mapping operations (standard OS interface) to implement virtual defragmentation

Avoids expensive physical copying of objects in the heap

Works well in practice because of dead object clustering