

# Compiler Construction: Practical Introduction

System Course for SRR Engineers

Samsung Research Russia  
Moscow 2019

# Lecture 7

## Runtime Support

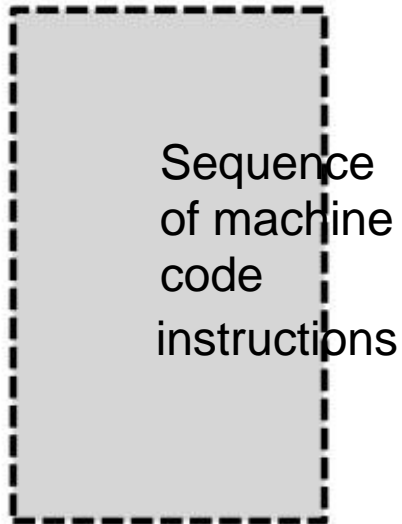
- C Memory Model
- C Memory Classes
- Run-time Stack & Activation Record
- Linux Process Memory Layout
- Garbage Collector
- Garbage Collector Algorithms
- GC in HotSpot JVM
- GC in .NET

# The C Memory Model

Each C program uses three kinds of memory:

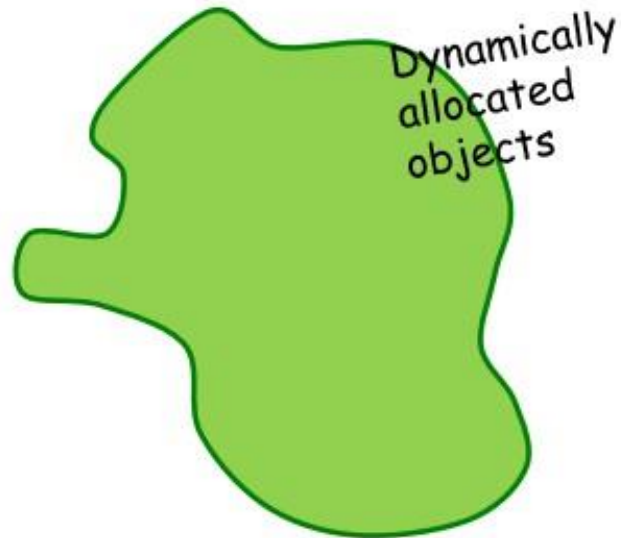
- Program
- Dynamic memory ("Heap")
- Stack

Program



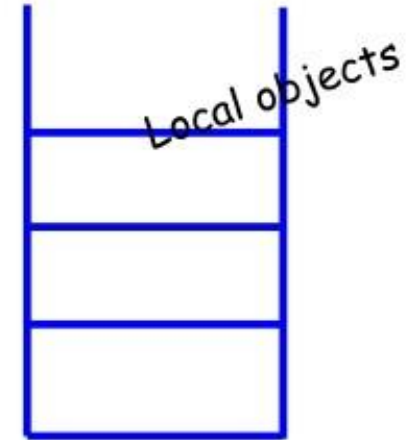
Program cannot modify this memory (self-modified programs are not allowed)

Heap



The discipline of using heap is defined by program **dynamic semantics**, i.e., at runtime (while program execution)

Stack



The discipline of using stack is defined by the (static) **program structure**

# C Global, Local & Dynamic Objects

## Global objects

Data Segments

Are created on program's start and exist ("live") until program is completed.

Live in the **global scope**.

Are accessible ("visible") within the translation unit they are declared in, OR within the whole program. May be initialized and uninitialized

## Local objects

Stack

Are created when a function is invoked or when the control flow enters a block, and disappear on return or on exit from the block.

## Dynamic objects

Heap

Are created and destroyed on arbitrary moments while program execution, following the program logic.

# Global, Local & Dynamic Objects

How global & local objects are created?

- By their declarations

How dynamic objects are created?

- Using special standard functions from the C library

**Globals & Locals:  
example**

```
int x;  
int* ptr;
```

`x` & `ptr` are **global objects**; they are created on the program's start and exist until its end

```
void f(int p)  
{  
    int* local = &x;  
    if ( p > 0 )  
    {  
        float m = .m;  
    }  
}
```

`p` & `local` are **local objects**; they are created when `f` function is invoked and disappear on return from `f`

`m` is the **local object**; it is created when the control flow enters the then-branch of `if` and disappears on return from this block

# Dynamic Objects

- How dynamic objects are created (and destroyed)?
- **Using special standard functions from the C library**

```
void* malloc ( int size )  
{  
Allocation algorithm  
}
```

```
void free ( void* ptr )  
{  
Deallocation algorithm  
}
```

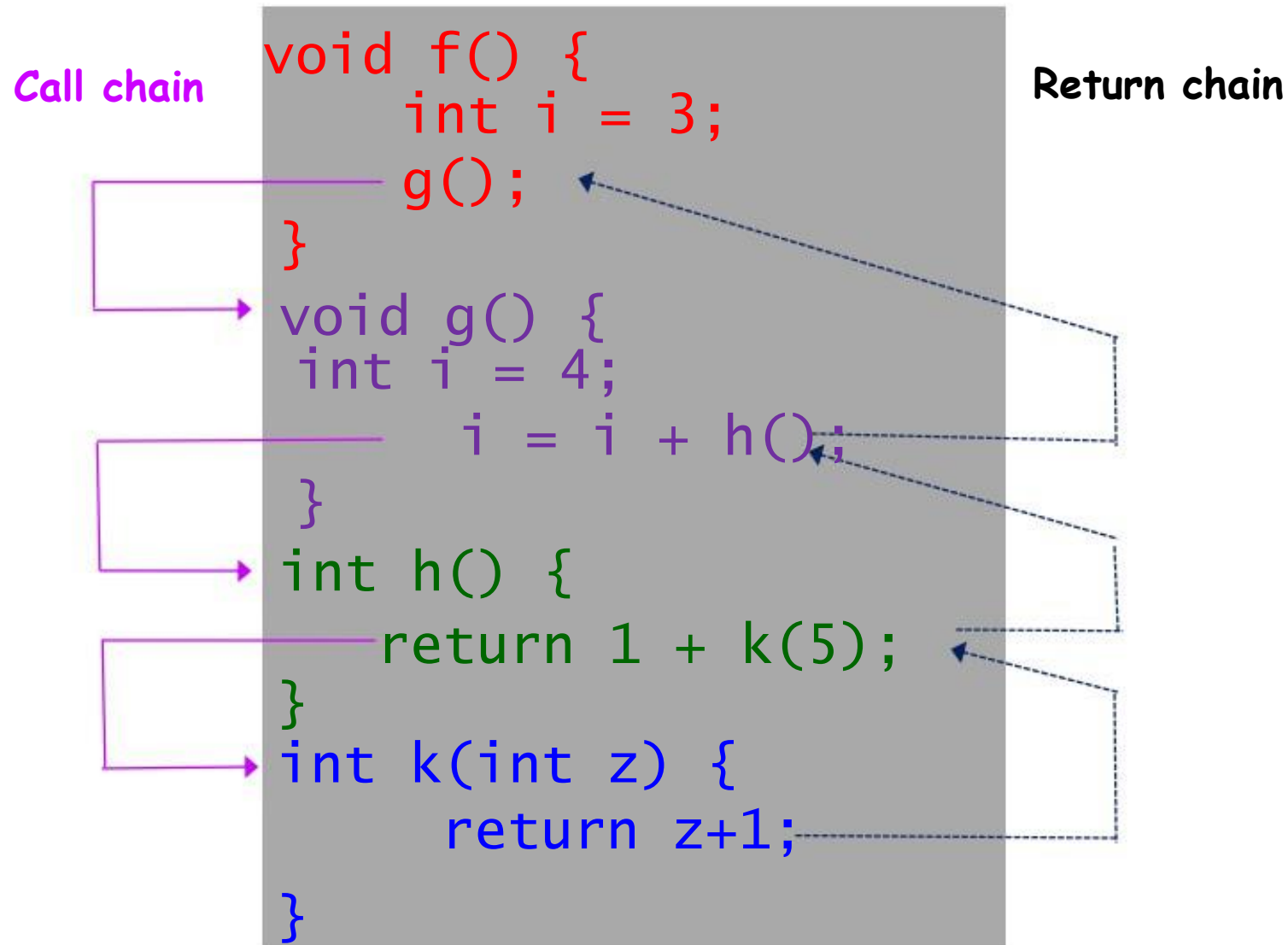
- Specification is a bit simplified.
  - The function allocates space for an object whose size (in bytes)  
is passed via the parameter.
  - The function returns a pointer to the memory allocated.
    - The pointer is "untyped" (`void*`).
  - There are more allocation functions in the library.

# Memory Model: Stack

- In C as well as in most modern languages the execution is centered around the **execution stack**.
- All algorithms are organized into **functions** (sometimes called **procedures**, **methods**, **routines** etc.)
  - The order of execution of functions is **LIFO**, i.e. the last function called is the first to terminate (this behavior is obtained using the stack)

Typical operations on stack:  
push, pop, empty

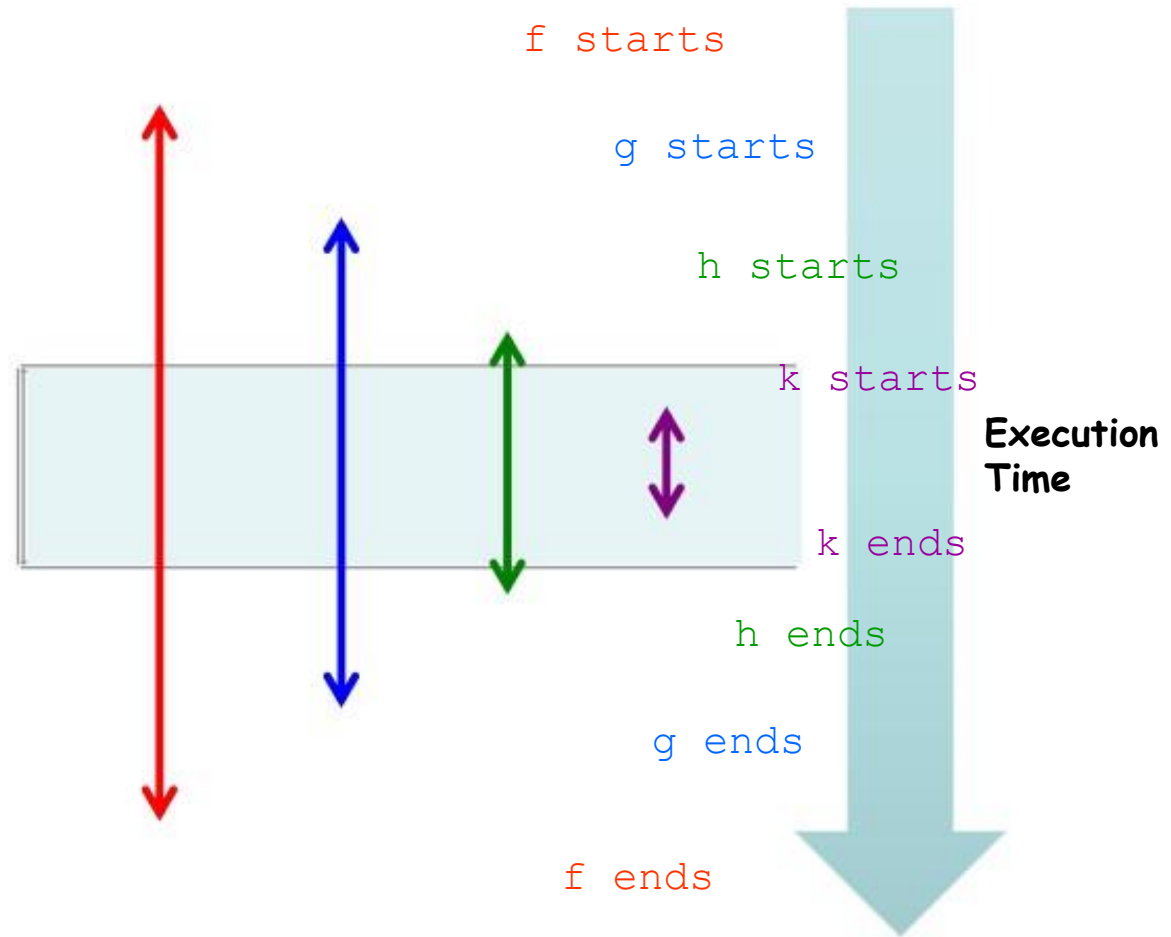
# Stack & Activation Record 1





# Stack & Activation Record 2

```
void f() {  
    int i = 3;  
    g();  
}  
void g() {  
    int i = 4;  
    i = i + h();  
}  
int h() {  
    return 1 + k(5);  
}  
int k(int z) {  
    return z+1;  
}
```



# Stack & Activation Record 3

- Each time a function is called, all the information specifically needed for the function execution are put on the stack
- That information is collectively called the **activation record (AR)** of the function call
- This allows recursion, since for each call there will be a separate activation record on the stack
  - When the call is completed (the function "returns") the corresponding AR is destroyed ("popped out" of the stack)
- Activation records are organized from bottom to top in memory diagram

# Stack & Activation Record 4

The information stored in the AR (also known as **Stack Frame**) for one call are the following:

- Information to restart the execution at the end of the call, i.e. after the function "returns"; these usually are:

- **Return address**

- **Pointer to the Stack portion devoted to the calling function**

- **Return value** (if any)

- Information needed to perform the computation (usually the **actual arguments** passed to the function in the call - if any)

- **Local variables** (if any)

## Stack & Activation Record 5

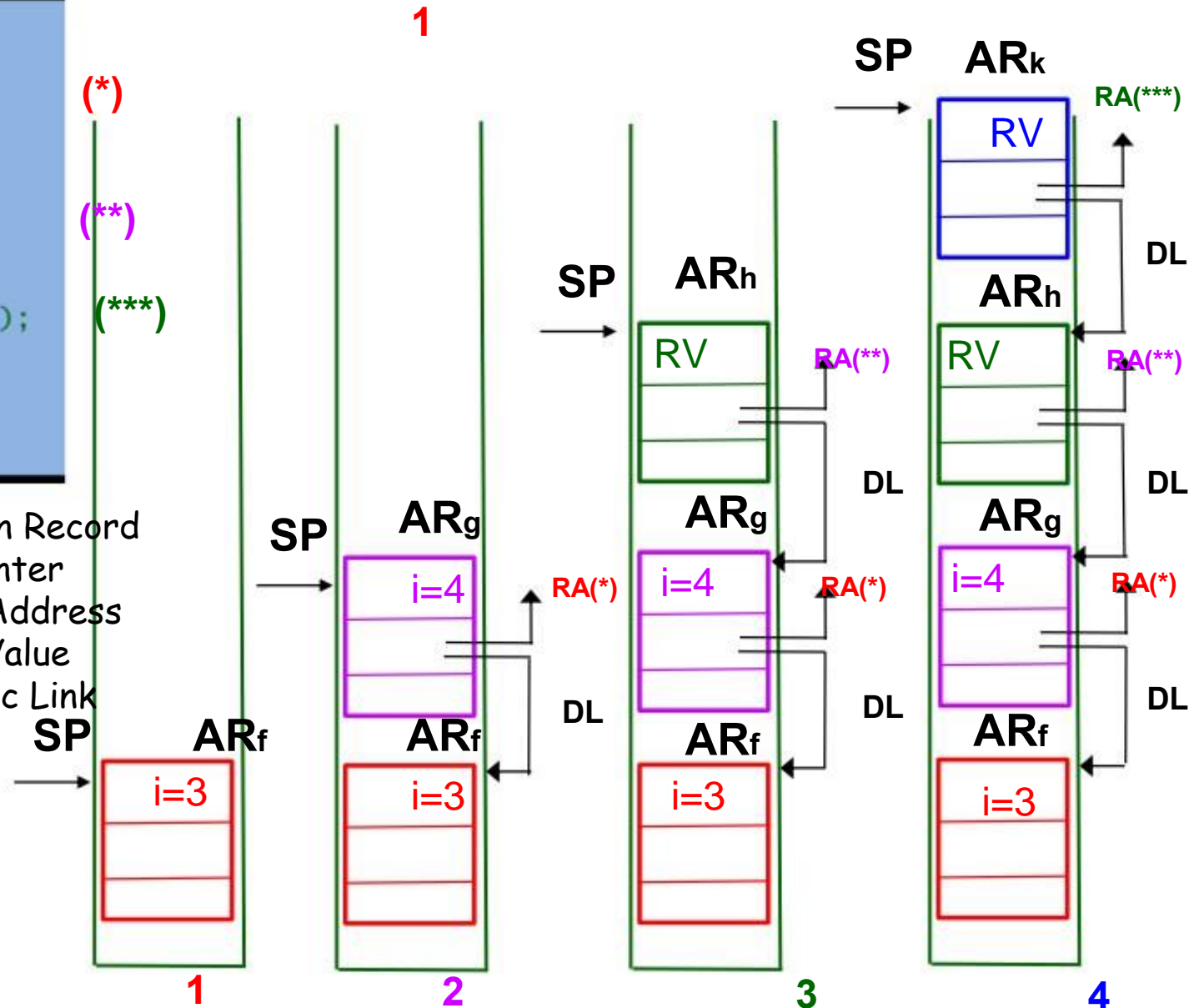
```
void f() {
    int i = 3;
    g();
}

void g() {
    int i = 4;
    i = i + h();
}

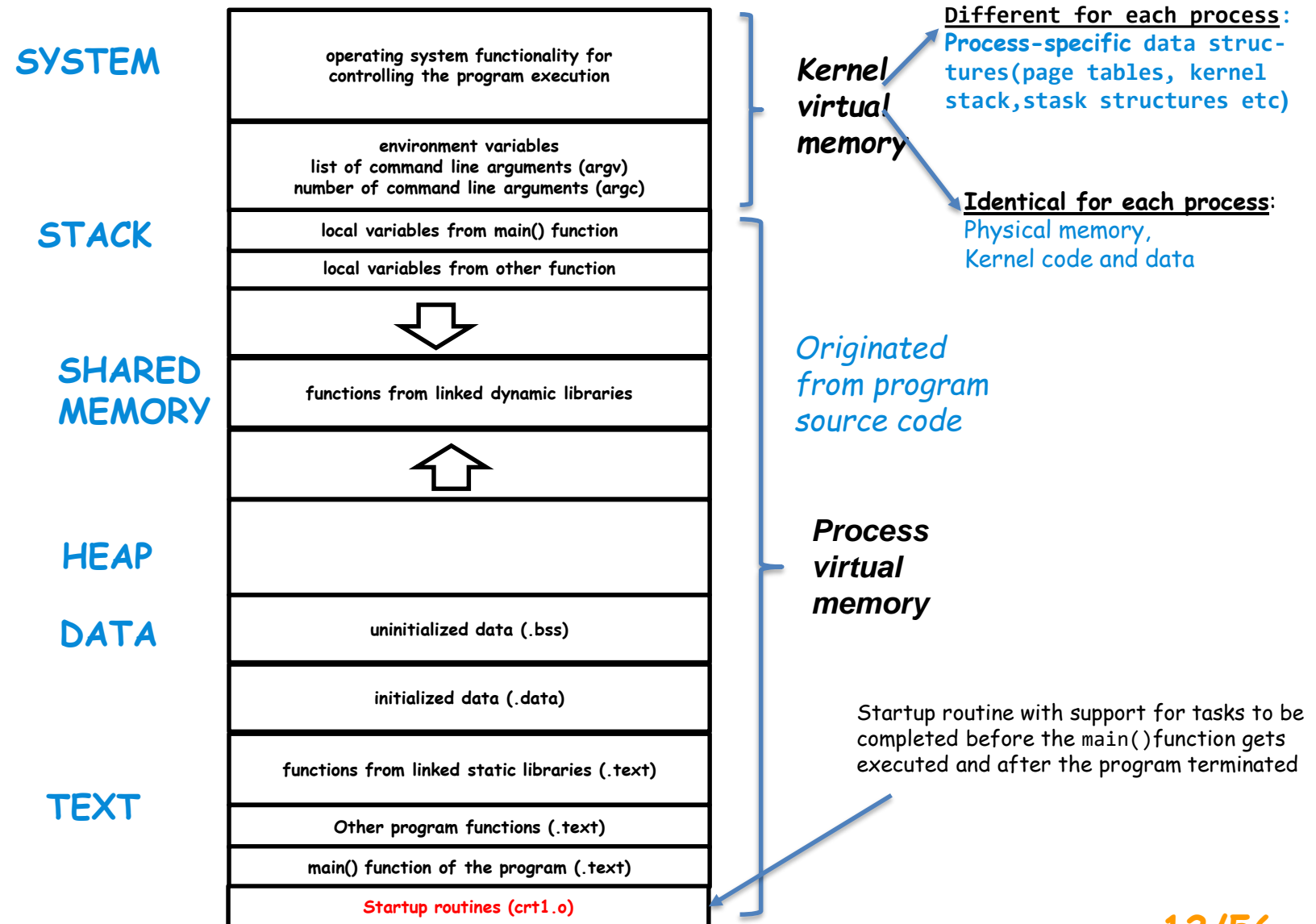
int h() {
    return 1 + k(5);
}

int k(int z) {
    return z+1;
}
```

- AR - Activation Record
- SP - Stack Pointer
- RA - Return Address
- RV - Return Value
- DL - Dynamic Link



# Linux Process Memory Map Layout



# What Is Garbage Collection?

- During program execution some data objects come up that cannot be accessed in the future.
- A program can have a memory flaw so it need to reclaim resources used by unaccessible objects.
- So, Garbage Collection is an automatic reclamation of computer memory storage.

# Garbage Collection: Pro & Con

- **Advantages:**

- Getting rid of dangling pointer bugs.
- Getting rid of double free bugs.
- Getting rid of certain type of memory leaks.

- **Disadvantages:**

- Requirement of extra computing resources.
- Unpredictable collection time.
- More memory-related work than useful work.

# Some Conclusions About GC

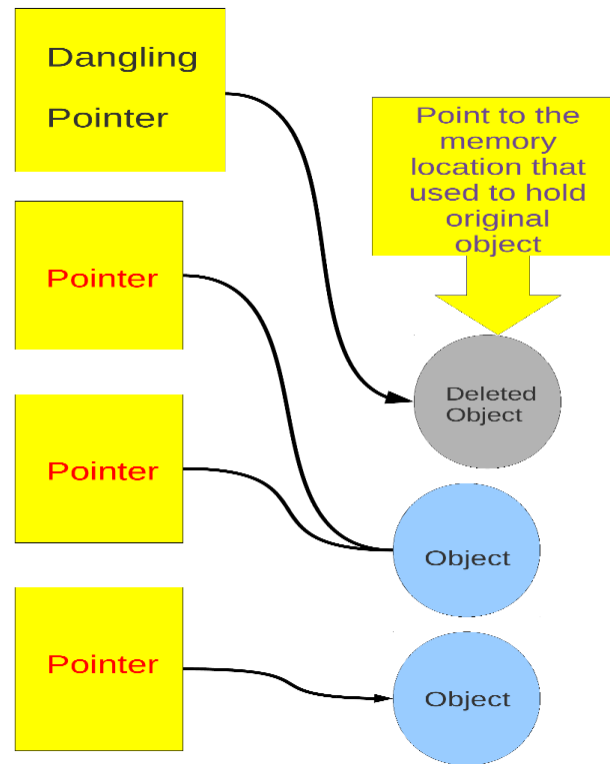
- Studies showed that performance of systems with well-implemented garbage collectors is highly competitive with systems with explicit deallocation.
- Automatic deallocation allows a programmer not to worry about memory management, increasing writeability of a system, and decreasing development time and costs.
- Explicit management introduces possibilities for making errors in memory management, and thus, decreases reliability.
- Garbage collection promotes purely modular design – explicit deallocation causes one module to be responsible for knowing that other modules are not interested in this particular object.



- Automatic Garbage Collectors:
  - First automatic garbage collection:
    - LISP (1958)
  - Several other languages implemented it:
    - BASIC (1964)
    - Logo (1967)
    - Java 1.0 (1996)

# GC Terms: Dangling Pointers

- Dangling pointer or wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type. These are special cases of memory safety violations. More generally, dangling references and wild references are references that do not resolve to a valid destination



# GC Terms: Memory Leak

- Memory leak in computer science is a type of resource leak that occurs when a computer program incorrectly manages memory allocation in such a way that memory which is no longer needed is not released. Memory leak may also happen when an object is stored in memory but cannot be accessed by the running code.

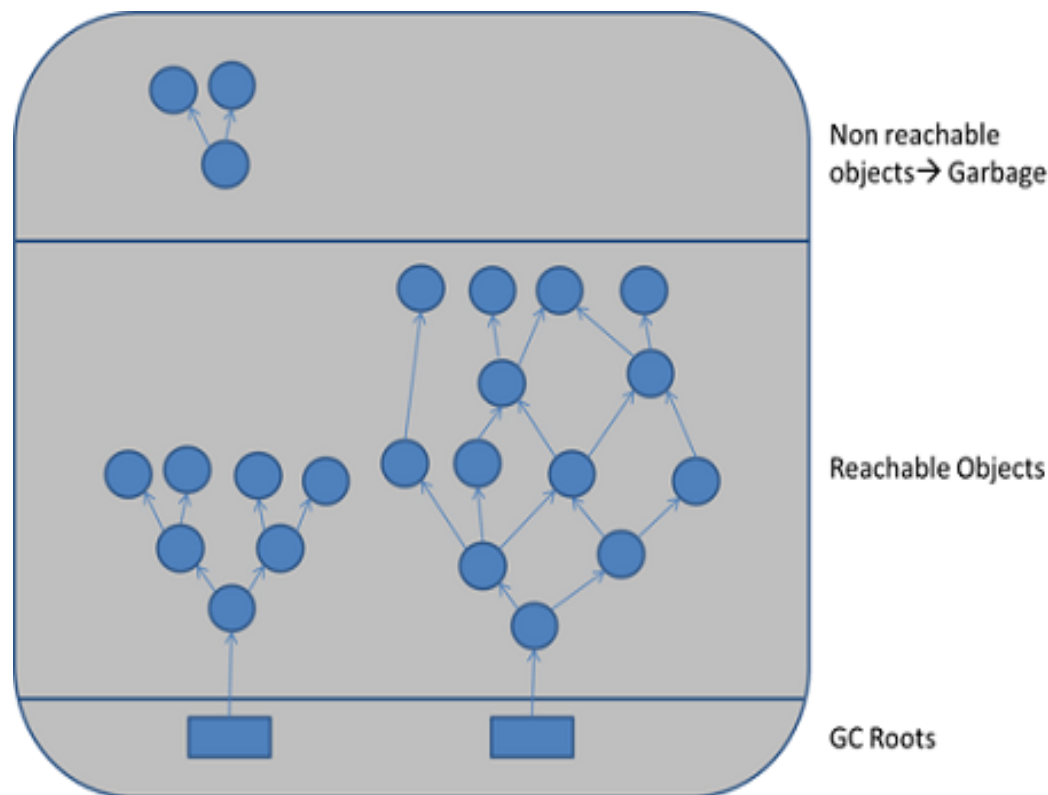
```
#include <stdlib.h>

void function_which_allocates(void) {
    /* allocate an array of 45 floats */
    float *a = malloc(sizeof(float) * 45);
    /* additional code making use of 'a' */
    /* return to main, having forgotten to free the memory we malloc'd */
}

int main(void) {
    function_which_allocates();
    /* the pointer 'a' no longer exists, and therefore cannot be freed,
       but the memory is still allocated. a leak has occurred. */
}
```

# GC Terms: GC Roots

- A **Garbage Collection Root** is an object that is accessible from outside the heap. Every object tree must have one or more root objects. As long as the application can reach those roots, the whole tree is reachable





# Examples of GC Roots

- **Local variables:** input parameters or locally created objects of methods that are still in the stack of a thread. For all intents and purposes, local variables are GC roots.
- **Active threads:** a started, but not stopped thread are always considered live objects and are therefore GC roots. This is especially important for thread local variables.
- **Thread locals:** thread local variables.
- **Static variables:** are referenced by their classes. This fact makes them de facto GC roots. Classes themselves can be garbage-collected, which would remove all referenced static variables.

# GC Roots for Simple Program

- A simple Java application has the following GC roots:
  - Local variables in the `main` method.
  - The `main` thread.
  - Static variables of the `main` class.

```
public class AddTwoNumbers {  
    static Number sum;   
  
    public static void main(String[] args) {  
        Number num1 = new Integer(5),  
              num2 = new Integer(15);   
        sum = new Integer (num1.intValue() + num2.intValue());  
        System.out.println("Sum of these numbers: "+sum.intValue());  
    }  
}
```

- **Generational:** collects young objects and long live objects separately.
- **Promotion:** allocation into old generation.
- **Marking:** finding all live objects.
- **Sweeping:** locating the dead objects.
- **Compaction:**
  - Defragments heap.
  - Moves objects in memory.
  - Remaps all affected references.
  - Frees contiguous memory regions.

- **Mutator:** user's program...
- **Parallel:** can use multiple CPUs.
- **Concurrent:** runs concurrently with program.
- **Pause:** a time duration in which the mutator is not running any code.
- **Stop-The-World (STW):** something that is done in a pause.
- **Monolithic:** something that must be done in it's entirely in a single pause.



# GC Terms: Collector Types

- **Parallel Collector:** uses multiple CPUs to perform Garbage Collection.
- **Concurrent Collector:** performs Garbage Collection work concurrently with the application's own execution.
- **Stop-The-World Collector:** performs Garbage Collection while the application is completely stopped.
- **Incremental Collector:** breaks the reachability analysis into smaller units. Mutator is executed between these units.

# GC Terms: Action Types

- **Mark (aka "Trace"):**
  - Starts from "roots" (thread stacks, statics, etc.).
  - "Paint" anything you can reach as "live".
  - At the end of **Mark** pass:
    - All reachable objects will be marked as "live".
    - All non-reachable objects will be marked as "dead" (aka "non-live").
  - **Note:** work is generally linear to "live sets".
- **Sweep:**
  - Scans through the heap, identify "dead" parts from "roots" (thread stacks, statics, etc.).
    - Usually in some form of free list.
  - **Note:** work is generally linear to heap size.

## ■ Compact:

- Over time, heap will get “swiss cheesed”: contiguous dead space between objects may not be large enough to fit new objects (aka “fragmentation”).
- Compaction moves live objects together to reclaim contiguous empty space (aka “relocate”).
- Compaction has to correct all object references to point to new object locations (aka “remap”).
- Remap scan must cover all references that could possibly point to relocated objects.
- **Note:** work is generally linear to “live sets”.

## ■ Copy:

- A copying collector moves all live objects from a "from" space to a "to" space & reclaim "from" space.
- At start of copy, all objects are in "from" space and all references point to "from" space.
- Start from "root" references, copy any reachable object to "to" space correcting references as we go.
- At end of copy, all objects are in "to" space, and all references point to "to" space.
- **Note:** work is generally linear to "live sets".

# GC Terms: Generations

- Generational Hypothesis: most objects die young.
- Focus collection efforts on young generation:
  - Use a moving collector: work is linear to the live set.
  - The live set in the young generation is a small & of the space.
  - Promote objects that live long enough to older generations.
- Only collect older generations as they fill up:
  - “Generational filter” reduces rate of allocation into older generations.
- Tends to be (order of magnitude) more efficient:
  - Great way to keep up with high allocation rate.

# Typical Combos In Commercial JVMs

- All notes below about **Server** variants.
- Young generation **usually** uses a copying collector.
- Young generation **usually** monolithic, stop-the-world.
- Old generation **usually** uses Mark/Sweep/Compact.
- Old generation may be STW, or Concurrent, or mostly-Concurrent, or incremental-STW, or mostly-incremental-STW.

# GC Algorithms: Mark-Sweep

- Mark-Sweep is usually run at specified time intervals.
- **Algorithm:**
  - **Step 1:** Starting from the root set, we trace through our graph of memory. Mark all objects reached.
  - **Step 2:** Sweep through memory and reclaim all unmarked space.

- **Pros:**

- The Mark-Sweep algorithm doesn't create drag on every single memory operation.

- **Cons:**

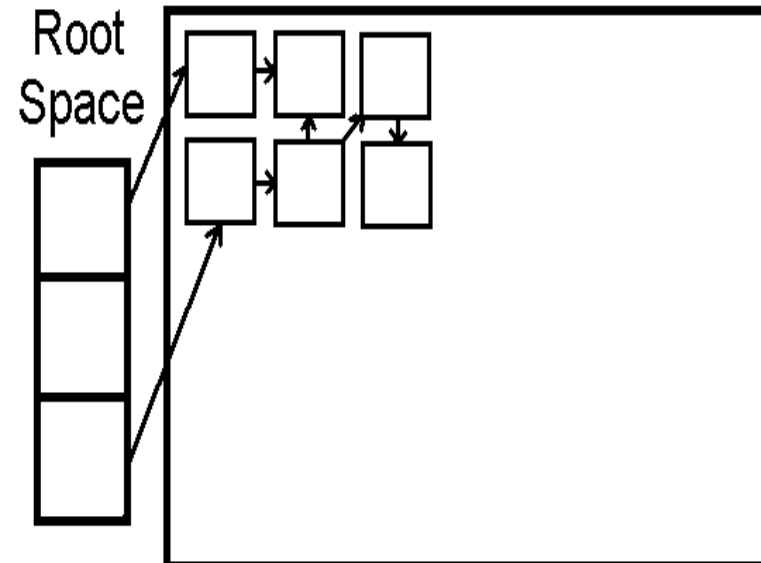
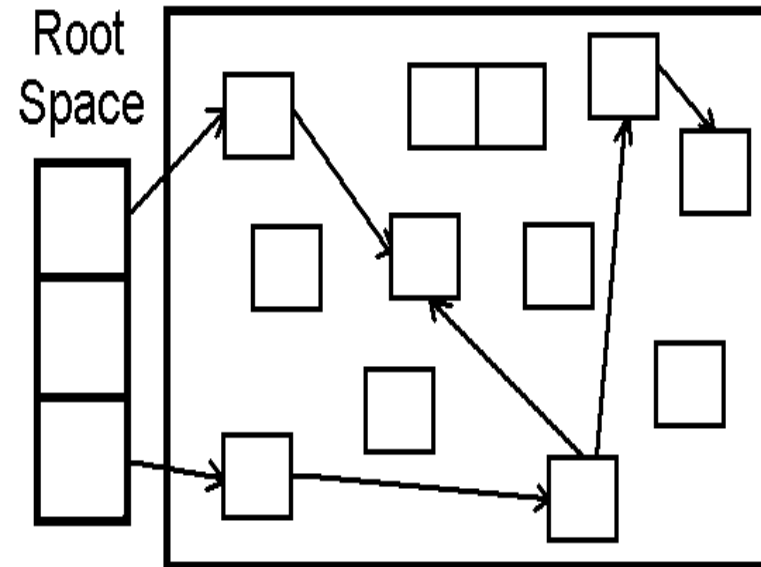
- Every location in memory must be examined during the sweep stage of this algorithm – this can be time consuming.
- Can leave several gaps in used memory when objects are swept out. This fragmentation of available memory can cause serious performance problems for applications which make heavy memory demands. Although in practice, this problem usually isn't a huge problem, Mark-Sweep garbage collection is usually considered unfit for high-performance systems for exactly this reason.



# GC Algorithms: Mark-Compact

- This algorithm is essentially a variation on the Mark-Sweep algorithm just described.
- **Algorithm:**
  - **Step 1:** All live objects in memory are marked, just as in Mark-Sweep.
  - **Step 2:** Instead of sweeping the dead objects out from under the live ones, the live objects are instead pushed to the beginning of the memory space. The rest of memory is reclaimed for future use.

# GC Algorithms: Mark-Compact



# GC Algorithms: Mark-Compact

- **Pros:**

- The fragmentation problem of Mark-Sweep collection is solved with this algorithm; available memory is put in a big single chunk.
- Also note that the relative ordering of objects in memory stays the same – that is, if object X has a higher memory address than Y before garbage collection, it will still have a higher address afterwards. This property is important for certain data structures like arrays.

- **Cons:**

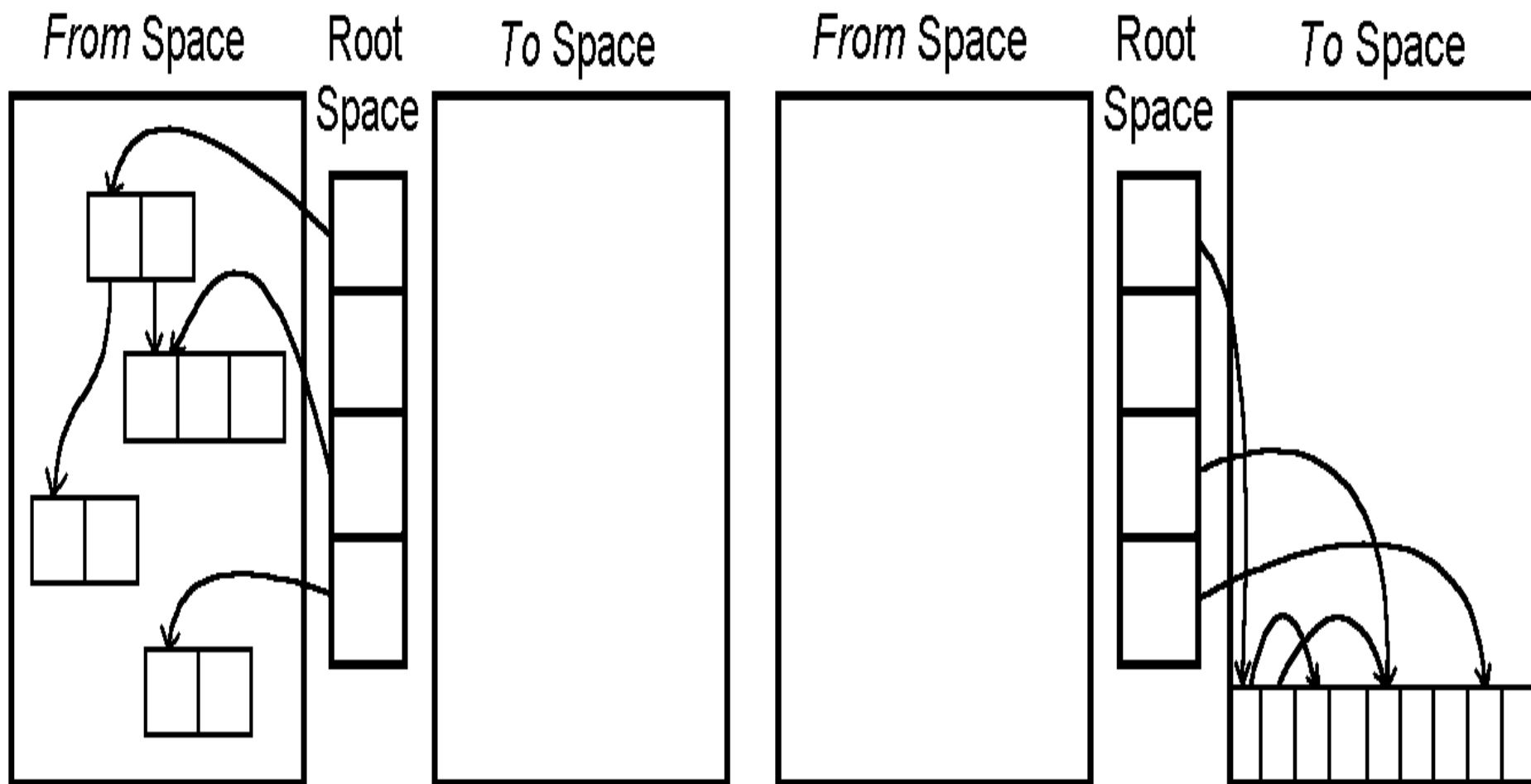
- The big problem with Mark-Compact collection is time. It requires even more time than Mark-Sweep collection, which can seriously affect performance.

- Like the Mark-Sweep algorithm, Copying garbage collection does not really collect garbage. The collector moves all live objects into an area of memory, so the rest of the heap is available to be used by the program since it contains garbage. This method integrates the copying process into the data traversal, so an object will only be visited once.

# GC Algorithms: Stop & Copy

- In this method the heap space is divided into two contiguous semi-spaces (from-space and to-space). During program execution, only one of these spaces is used.
- Memory is allocated linearly upwards in the current semi-space as demanded by the execution program. When the space is exhausted the program is stopped and the garbage collector is executed.
- All live objects are copied from the current semi-space to the other semi-space. The roles of the two semi-spaces are reversed each time the garbage collector is invoked.

# GC Algorithms: Stop & Copy



# GC Algorithms: Generational

- One of the limitations of simple garbage collection algorithms is that the system has to analyze all the data in heap. For example, a Copying Algorithm has to copy all the live data every time it is used. This may cause significant increases in execution time.
- Studies in 1970s and 1980s found that large Lisp programs were spending from 25 to 40 percent of their execution time for garbage collection.
- Other studies show that most objects live for very short time (so-called “weak generational hypothesis”), so most objects have to be deallocated during the next garbage collection.
- The opposing theory, the “strong generational hypothesis”, which states that the older an object is, the more likely it is to die, does not appear to hold. Object lifetime distribution does not fall smoothly, and if an object has survived a few collections, it is likely to live quite long.

# GC Algorithms: Generational

- **Implication:** if we can concentrate on collection of young objects and do not touch too often older ones, the amount of data that has to be analyzed and copies is considerable reduced. We can therefore make significant gains in garbage collection efficiency.
- This approach, which allows us to avoid analyzing older objects during each collection (this keeping the costs of collection down), is called **Generational Collection**.

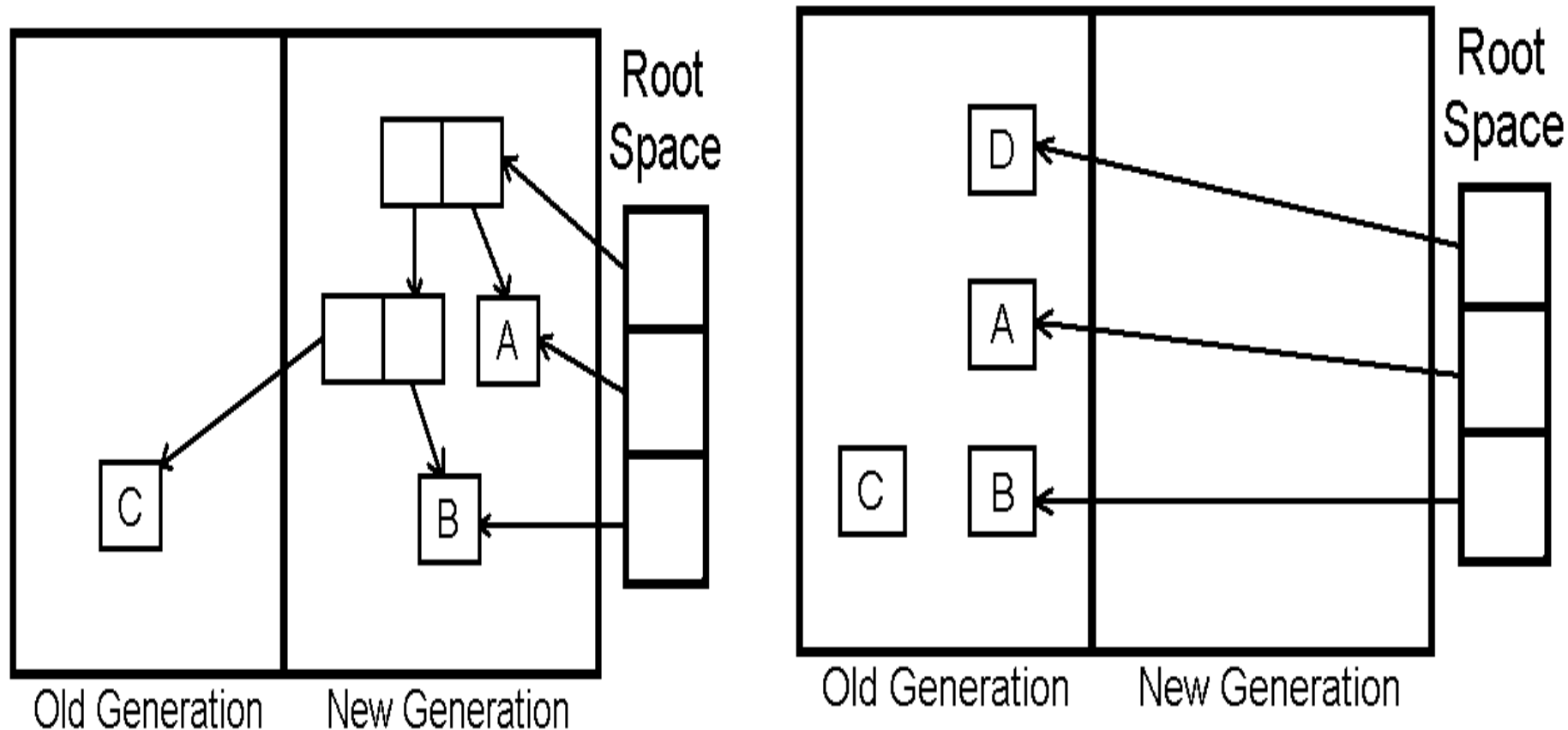


## ■ How does it work? :

- Generational garbage collection divides the heap into two or more regions, called **generations**.
- Objects are always allocated in the youngest generation.
- The garbage collection algorithm scans the youngest generation most frequently, and performs scanning of successive generation more rarely.
- Most objects in youngest generation are deallocated during the next scan. However, those objects that survive a few scans or reach a certain age are advanced to the next generation.

# GC Algorithms: Generational

- How does it work? :



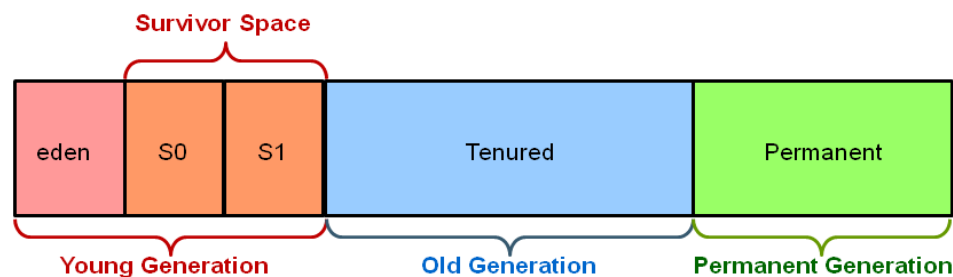
- **Difficulties with Generational Collection:**
  - In order for Generational Collection to work, it must be possible to collect data in younger generations without collecting the older ones.
  - This leads to some problems: if there exists a pointer from object2 in the older generation to object1 in the younger, object1 should be obviously considered alive.
  - So, generational collection algorithms should check whether there are any pointers from objects stored in one generation to objects in other, and record inter-generational pointers from older generations to younger ones.

- Difficulties with Generational Collection :
  - Such pointers may arise in two situations:
    - An object containing a pointer is promoted to older generation.
    - The pointer is directly stored in the memory.
  - In the first case, inter-generation pointers can be easily recorded by checking each object during its promotion. The second case is harder - the collector needs to check each pointer store and provide some extra bookkeeping in case an inter-generational pointer is created. The process of trapping pointer stores and recording them called "write barrier".
  - **Overall:** generational collection significantly improves the performance of collectors for most of programs. Such collectors are in widespread use.

# GC in HotSpot JVM: Generations

- The heap is broken into smaller parts of generations:
  - Young Generation.
  - Old or Tenured Generation.
  - Permanent Generation.

Hotspot Heap Structure



# GC in HotSpot JVM: Generations

- **Permanent Generation:**

- Contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.
- Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

- **Young Generation:**

- Is where all new objects are allocated and aged. When the young generation fills up, this causes a **minor garbage collection**. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

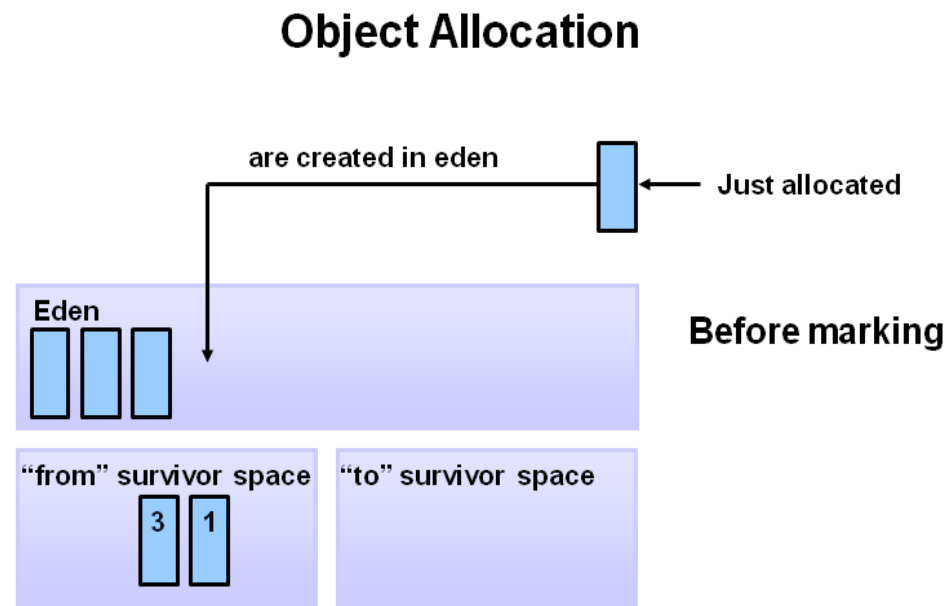
# GC in HotSpot JVM: Generations

- **Old Generation:**

- Is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a **major garbage collection**.
- **Major garbage collection** are also Stop-The-World events. Often a major collection is much slower because it involves all live objects. So for responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop-The-World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

# GC in HotSpot JVM: Allocation Phases

- First, any new objects are allocated to the eden space. Both survivor spaces start out empty.

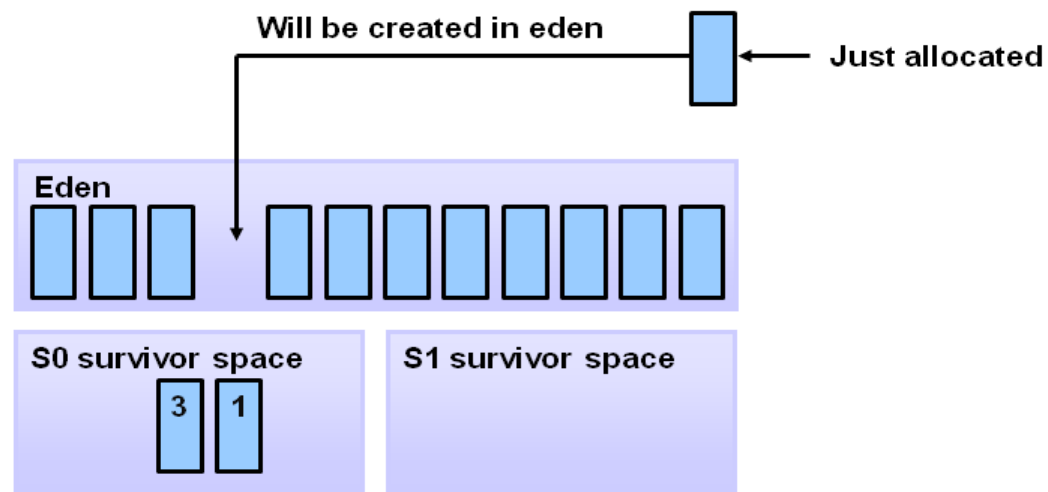




# GC in HotSpot JVM: Allocation Phases

- When the eden space fills up, a minor garbage collection is triggered.

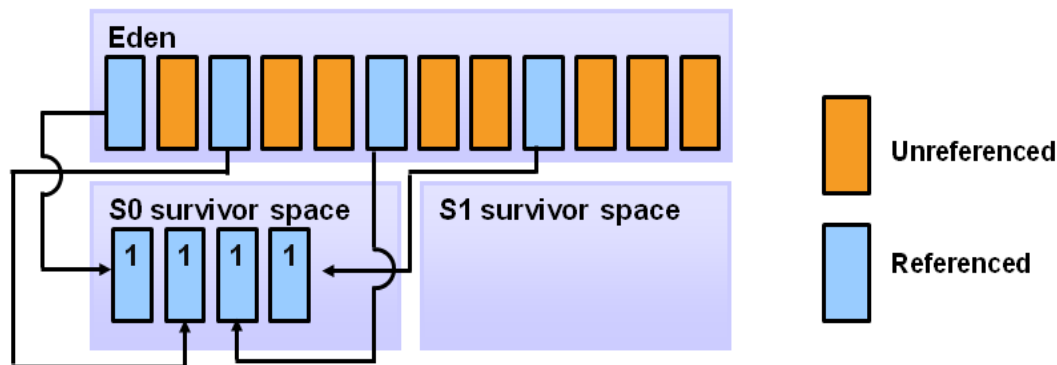
## Filling the Eden Space



# GC in HotSpot JVM: Allocation Phases

- Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is clear.

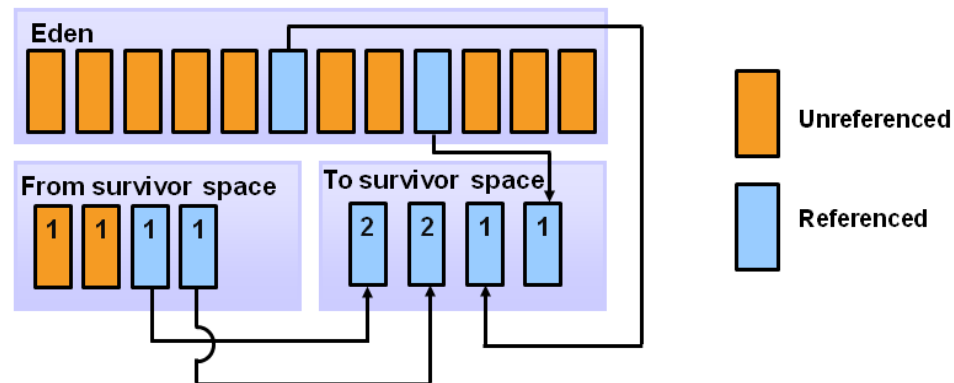
## Copying Referenced Objects



# GC in HotSpot JVM: Allocation Phases

- At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to the next survivor space (S1), as the objects from S0 with their age incremented. Both S0 and eden are cleared.

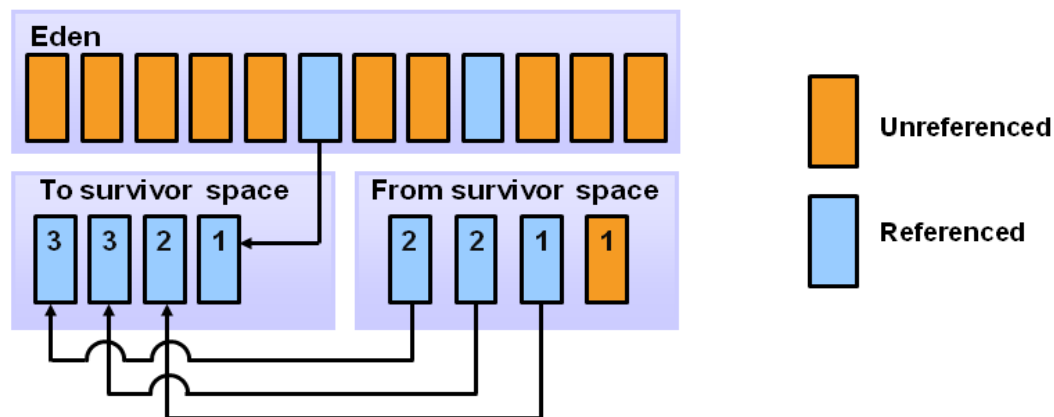
## Object Aging



# GC in HotSpot JVM: Allocation Phases

- At the next minor GC, the same process repeats with the survivor spaces switch: Referenced objects are moved to S0, surviving objects are aged, eden and S1 are cleared

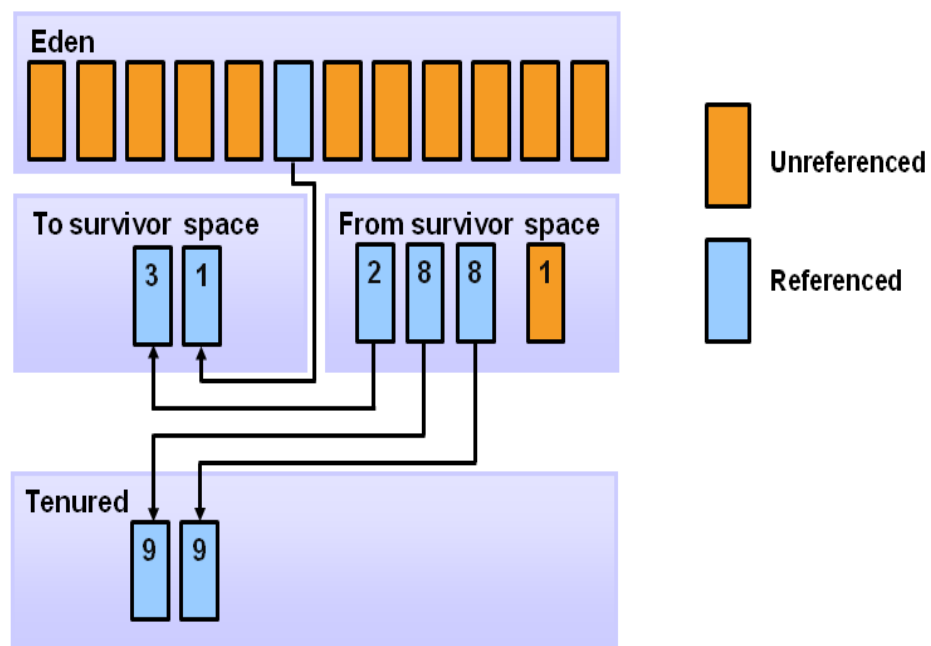
## Additional Aging



# GC in HotSpot JVM: Allocation Phases

- When aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.

## Promotion



# GC in HotSpot JVM: Collector Types

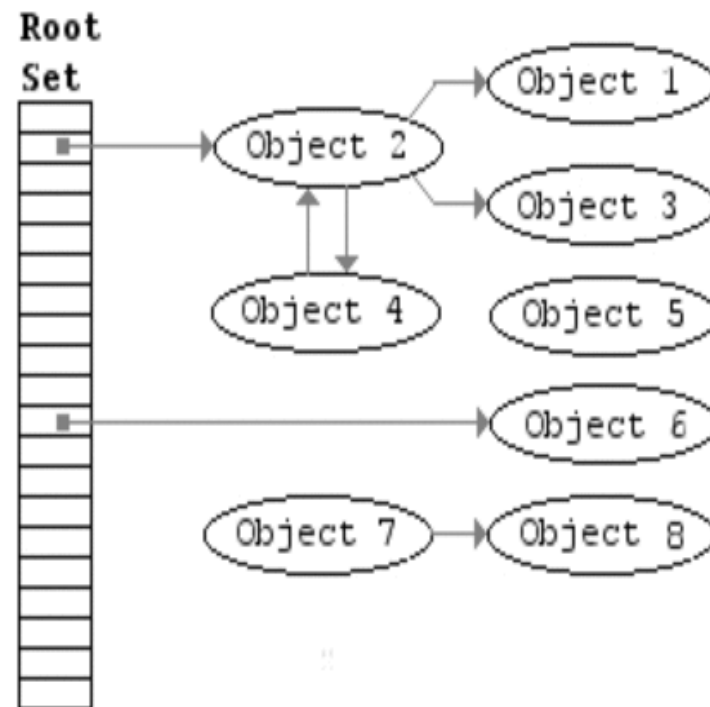
- Oracle HotSpot JVM has four types of GC:
- **Serial Garbage Collector**: the simplest implementations that works with a single thread, it freezes all application threads. It is for using only for client applications.
- **Parallel Garbage Collector (Throughput Collector)**: the default GC uses multiple threads for managing heap space, also freezes other application threads.

# GC in HotSpot JVM: Collector Types

- **CMS Garbage Collector (Concurrent Mark Sweep):** uses multiple threads for garbage collection and shares processor resources with the application threads. Applications using this type of GC respond slower on average but do not stop responding to perform garbage collection.
- **Garbage First (G1) Collector:** designed for applications running on multiple-processor machines with large memory space. Unlike other collectors, *G1 collector* partitions the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory. When performing garbage collections, *G1* shows a concurrent global marking phase (i.e. phase 1 known as Marking) to determine the liveness of objects throughout the heap. After the mark phase is completed, *G1* knows which regions are mostly empty. It collects in these areas first, which usually yields a significant amount of free space (i.e. phase 2 known as Sweeping). It is why this method of garbage collection is called *Garbage-First*.

# GC in .NET: Algorithm

- .NET and CLR are making use of tracing garbage collector. That means that on every collection the collector figures out whether the object is used by tracing every object from stack roots, GC handles and static data.
- Every object that could be traced is marked as live and at the end of tracing the ones without the mark are removed (swept) from the memory and it gets compacted. This is called a simple **mark and sweep algorithm**.

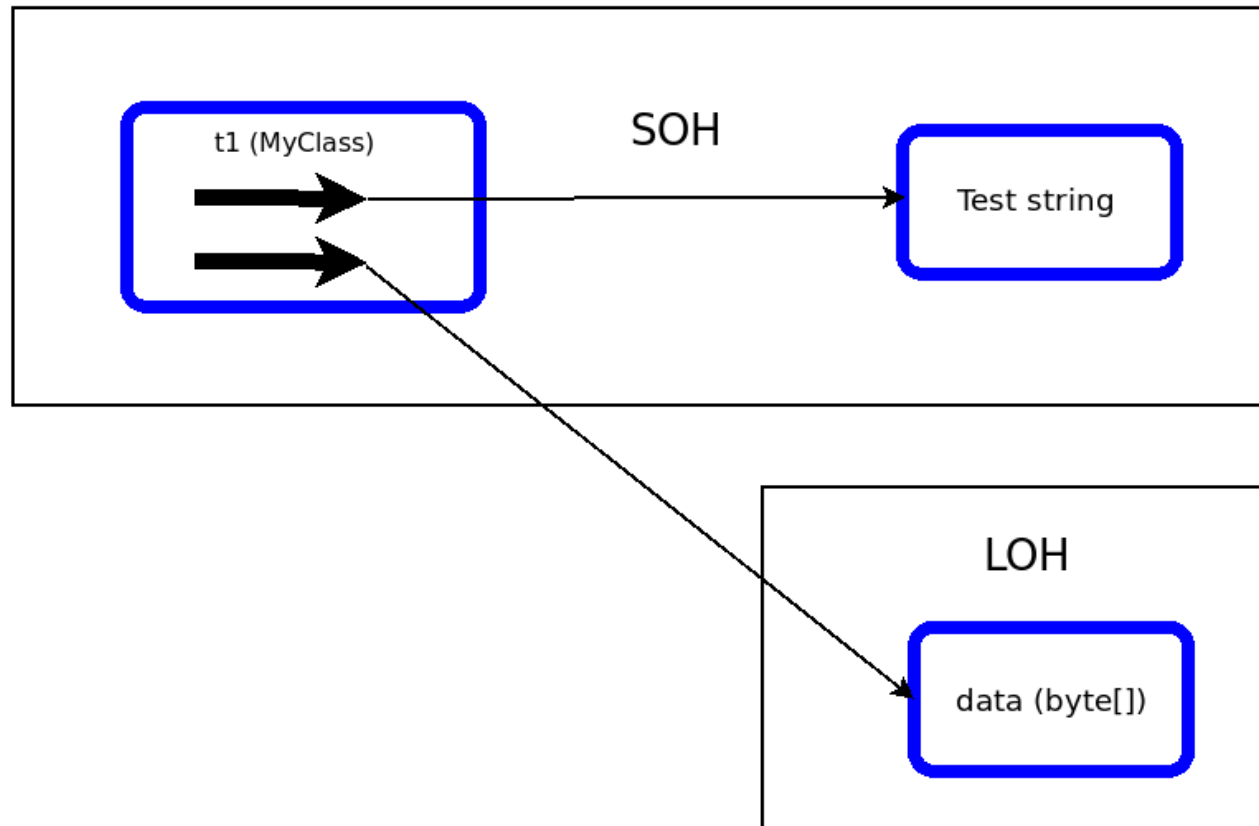




- Stack references.
- Global/static object references.
- CPU registers.
- Object finalization references.
- Interop(ability) (unmanaged) references (.NET object passed to COM/API calls)

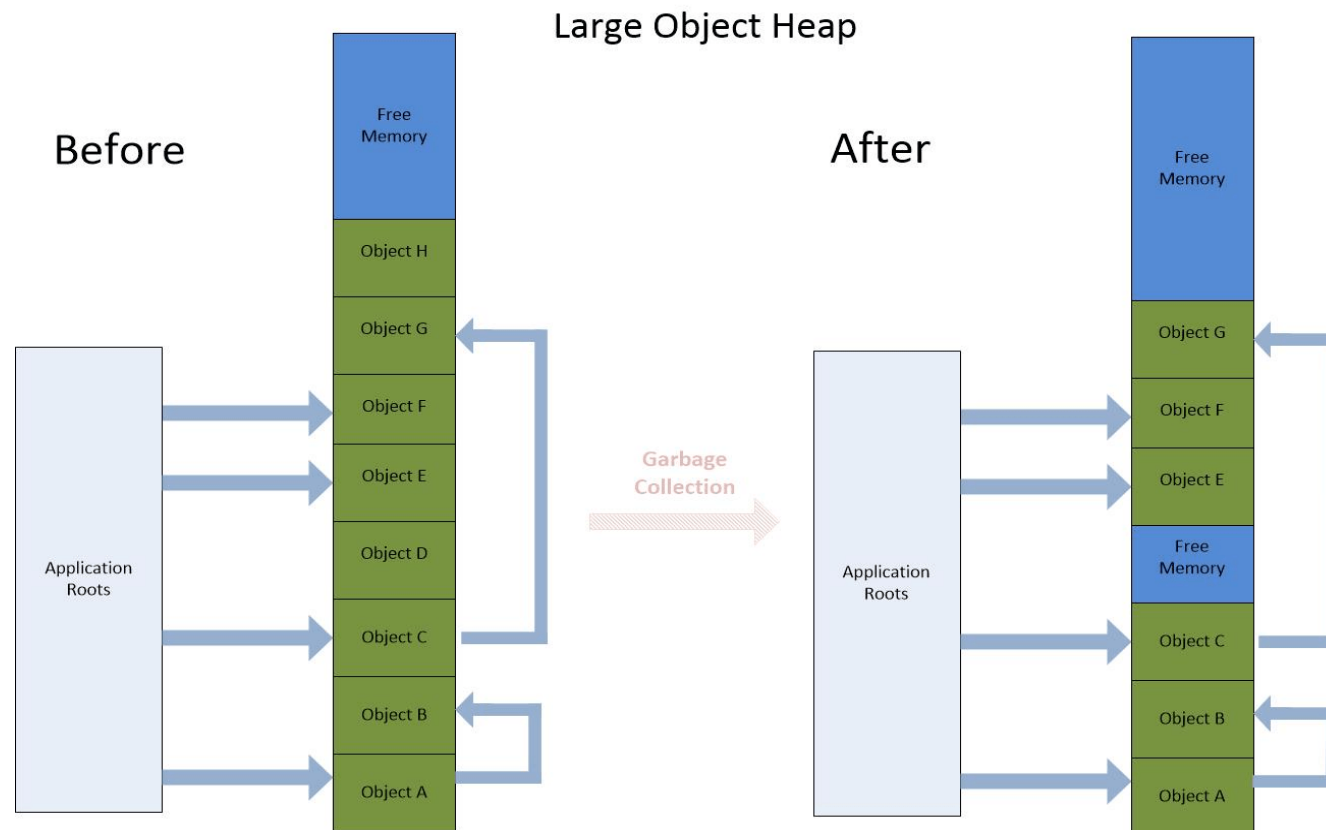
# GC in .NET: Managed Object by Size

- Small Object Heap (SOH) - objects smaller than 85 Kb.
- Large Object Heap (LOH).



# GC in .NET: Large Objects Heap (LOH)

- The primary difference is that the garbage collector **never compacts** this heap. This increase performance, but there may not be enough room for a new object even with enough available memory. This causes the Common Language Runtime (CLR) to throw `OutOfMemoryException`.



# GC in .NET: Small Objects Heap (SOH)

- The GC uses three segments, called **generation**, for small objects:
  - Generation 0 “short-live-generation”, new objects are placed here.
  - Generation 1 “generation-in-between”.
  - Generation 2 “long-live-generation” gets collected less often. GC over it is full garbage collection.

