

# Garbage Collection

*Instructor: Mainack Mondal*

*Scribe-By: Basa Sreekar*

## Outline:

In this lecture, we will cover Memory Leaks, Garbage Collection, and Reference Counting. A memory leak is a type of resource leak caused by incorrect memory management of dynamic program memory. Garbage Collection (GC) is an automatic memory management feature that deals with unreferenced objects. This is a built-in feature in languages like C#, Java, and Python. GC algorithms reclaim memory in the heap that is no longer useful, and some may also manage efficient allocation of heap memory. Reference counting is the simplest GC technique, which keeps track of the number of references to heap objects and clears them when they are unreferenced.

## Contents

<b>1</b>	<b>Memory Structure and Memory Leaks</b>	<b>2</b>
1.1	Memory Layout: . . . . .	2
1.2	Memory Leaks . . . . .	3
1.2.1	Smart pointers . . . . .	3
1.2.2	Valgrind . . . . .	5
<b>2</b>	<b>Garbage Collection</b>	<b>10</b>
2.1	Cells and Liveness . . . . .	10
<b>3</b>	<b>Garbage Collection Techniques</b>	<b>10</b>
3.1	Reference Counting . . . . .	10
3.2	Disadvantages of Reference Counting . . . . .	10

# 1 Memory Structure and Memory Leaks

## 1.1 Memory Layout:

Program memory is broadly classified into three sections.

1) Command Line Arguments and Environmental variables.

### 2) Static Memory

- Text/Code segment stores the executable instructions.
- Initialized Data Segment stores static and global variables which are initialized.
- Uninitialized Data Segment stores static and global variables that are declared but not initialized.

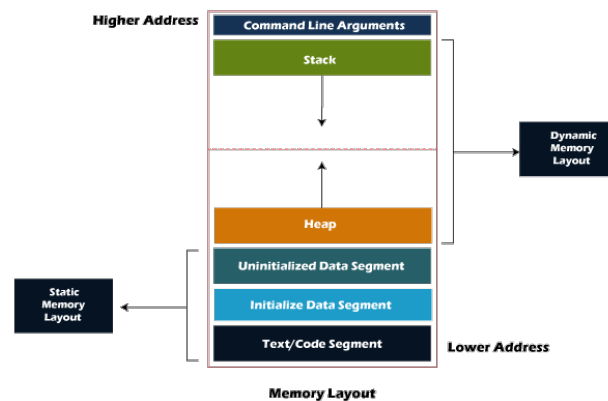


Figure 1: Memory Layout in C

### 3) Dynamic Memory

#### Stack:

- This section stores function parameters and local variables.
- The allocation happens on contiguous blocks of memory.
- Follows Last in First out (LIFO) principle, which allows the function calls to return to their previous state.
- Due to LIFO structure, scope of variables is defined properly and can be cleared if scope ends.

#### Heap:

- This section where memory is allocated dynamically at runtime.
- Usually heap address grows from low to high, while stack address grows from high to low.
- Heap memory allocation is often non-contiguous.
- So proper handling of memory by the programmer is expected! or it may result in memory leak.

## 1.2 Memory Leaks

- As mentioned earlier memory leak is a type of resource leak.
- It causes degraded performance and bad user experience.
- Memory leak is not an immediate issue, it doesn't raise any error while running a program, but it slowly eats memory and may leads to system crash.
- This makes memory leaks hard to notice,so special tools are required to detect them.
- Also memory leak is not a permanent problem, so there are solutions to this problem.

**Manual Memory Management:** Here programmer should implement this and prone to mistakes. Prevention techniques:

- Using STL containers in C++ like 'vector', 'string', 'map' ..etc , they automatically manage memory.
- Using smart pointers 'std::unique\_ptr', 'std::shared\_ptr' and 'std::weak\_ptr', to ensure automatic deallocation.
- Using memory detecting tools like valgrind and Sanitizers, which can't prevent, but checks for memory-related issues in the code.

### 1.2.1 Smart pointers

- Smart pointers ensure that the object is deleted when it is no longer used. They are like wrappers around the normal pointers.
- The 'auto\_ptr' was the first implementation of a smart pointer in the standard library, after c++11 it was replaced.
- They are part of the C++ Standard Library and are defined in the 'memory' header.

#### **Unique pointer:**

- 'std::unique\_ptr' is used when a dynamically allocated object needs a single owner.
- Thus they have exclusive ownership on the object so that they can't be copied and it will be deleted if goes out of its scope.

#### **Shared pointer:**

- 'std::shared\_ptr' Shared pointer works based on reference counting technique.
- These are used when multiple pointers need to refer the same object (Shared ownership), copying the pointer increases reference count.
- The object is deleted only when the last shared pointer owning it is deleted.
- Function '.use\_count()' returns how many shared pointers owns the object. Here are some basic code to understand properties of different smart pointers.

**Example code for Smart Pointers:**

```
1  #include <iostream>
2  #include <string>
3  #include <memory>
4  using namespace std;
5
6  class check{
7      public:
8          check() { cout<<"Created check"<<endl; } //constructor
9          ~check(){ cout<<"Destroyed check"<<endl; } //destructor
10         string function() { return "Acessing a class using unique pointer";}
11 };
12
13 int main()
14 {
15     cout<<"unique pointer : "<<endl;
16     {
17         //unique_ptr<check> test(new check()); // not suggested
18         unique_ptr<check> test = make_unique<check>(); //creating a unique pointer
19         cout<<test->function()<<endl; // calling function of the class
20     }
21     // cout<<test->function()<<endl; // use this to verify test scope
22     //unique_ptr<check> test1 = make_unique<check>();
23     cout<<endl;
24     cout<<"shared pointer : "<<endl;
25     shared_ptr<check> shrd1;
26     {
27         shared_ptr<check> shrd2 = make_shared<check>();
28         cout<<"Reference count = "<<shrd2.use_count()<<endl;
29         cout<<"Shairng reference of shared pointer to other shared pointer"<<endl;
30         shrd1 = shrd2;
31         cout<<"Reference count = "<<shrd2.use_count()<<endl;
32     }
33     cout<<endl;
34     cout<<"weak pointer : "<<endl;
35     weak_ptr<check> weak;
36     {
37         shared_ptr<check> shrd = make_shared<check>();
38         cout<<"Reference count = "<<shrd.use_count()<<endl;
39         cout<<"Shairng reference of shared pointer to a weak pointer"<<endl;
40         weak = shrd;
41         cout<<"Reference count = "<<shrd.use_count()<<endl;
42     }
43     cout<<endl;
44     return 0;
45 }
```

**Output:**

unique pointer :

Created check

Accessing a class using unique pointer

Destroyed check

shared pointer :

Created check

Reference count = 1

Shairng reference of shared pointer to other shared pointer

Reference count = 2

weak pointer :

Created check

Reference count = 1

Shairng reference of shared pointer to a weak pointer

Reference count = 1

Destroyed check

Destroyed check

**Weak pointer:**

- Weak pointers ('std::weak\_ptr') are used to break circular references that could lead to memory leaks.
- Weak pointer doesn't own the object, it is only a weak reference to an object which is managed by a shared pointer.
- Using 'lock()' function a weak pointer can be converted to shared pointer, if access to the object exists (this can be checked using '.expired()' function).

**1.2.2 Valgrind**

- Valgrind is a memory debugging and profiling tool.
- Valgrind is used to detect memory errors and memory leaks.
- The most popular and default tool in valgrind is 'Memcheck', which detect issues like illegal memory access, use of uninitialized data, memory leaks etc.
- Also 'callgrind' profiling tool is used to analyse program's call graph and instructions.
- 'Massif' tool is used to measure the heap memory usage of the program.
- Massif tracks heap memory usage by taking snapshots at various points during the program's execution, showing where and how memory is allocated, but not whether this memory was released later in the program.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void f1(){
5      const int a = 15;
6      int* arr = malloc(sizeof(*arr)*a); //dynamic memory allocation
7      printf("Dynamically allocated size in f1 = %ld\n",sizeof(*arr)*a);
8      for(int i=0;i<a;i++){
9          arr[i] = i*2+1;
10         printf("arr[%d] = %d\n",i,arr[i]);
11     }
12     // problem 1: memory leak -- arr not freed
13 }
14
15 void f2(void){
16     const int b = 10;
17     int* x = malloc(b * sizeof(*x));
18     printf("Dynamically allocated size in f2 = %ld\n",sizeof(*x)*b);
19     x[10] = 0; // problem 2: heap block overrun, this should be highlighted by valgrind tools.
20     free(x); // keep(or)comment this line, to get two cases
21     // problem 3: memory leak -- x not freed, if we comment line 20
22 }
23
24
25 // We expect valgrind to highlight the above problems
26 int main(){
27     f1();
28     f2();
29     return 0;
30 }
```

- Now using valgrind `-leak-check=full ./filename` gives detailed memory analysis
- While using 'massif' if memory usage only increases and does not drop, and program's logic should be freeing that memory, then it might be indicative of a leak.

### Case1: Commenting line 20

- 'definitely lost: 100 bytes in 2 blocks:' This means that 100 bytes of memory were allocated and not freed.
- These blocks are "definitely lost" because the program no longer has any references to this memory, meaning they are true memory leaks.
- Here leak is due to f1 and f2 functions, 'total heap usage: 3 allocs, 1 frees, 1,124 bytes allocated:' shows the program made three memory allocation requests and only freed, leaving the others still allocated.

## Garbage Collection

```
sreekar@sreekar-IdeaPad:~/Documents/cpp$ valgrind --leak-check=full ./Valgrind
==49694== Memcheck, a memory error detector
==49694== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==49694== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==49694== Command: ./Valgrind
==49694==
Dynamically allocated size in f1 = 60
arr[0] = 1
arr[1] = 3
arr[2] = 5
arr[3] = 7
arr[4] = 9
arr[5] = 11
arr[6] = 13
arr[7] = 15
arr[8] = 17
arr[9] = 19
arr[10] = 21
arr[11] = 23
arr[12] = 25
arr[13] = 27
arr[14] = 29
Dynamically allocated size in f2 = 40
==49694== Invalid write of size 4
==49694== at 0x109288: f2 (in /home/sreekar/Documents/cpp/Valgrind)
==49694== by 0x109287: main (in /home/sreekar/Documents/cpp/Valgrind)
==49694== Address 0x4a7f528 is 0 bytes after a block of size 40 alloc'd
==49694== at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-and64-linux.so)
==49694== by 0x109238: f2 (in /home/sreekar/Documents/cpp/Valgrind)
==49694== by 0x109287: main (in /home/sreekar/Documents/cpp/Valgrind)
==49694==
==49694== HEAP SUMMARY:
==49694==   in use at exit: 100 bytes in 2 blocks
==49694== total heap usage: 3 allocs, 1 frees, 1,124 bytes allocated
==49694==
==49694== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==49694== at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-and64-linux.so)
==49694== by 0x109238: f2 (in /home/sreekar/Documents/cpp/Valgrind)
==49694== by 0x109287: main (in /home/sreekar/Documents/cpp/Valgrind)
==49694==
==49694== 60 bytes in 1 blocks are definitely lost in loss record 2 of 2
==49694== at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-and64-linux.so)
==49694== by 0x10918C: f1 (in /home/sreekar/Documents/cpp/Valgrind)
==49694== by 0x109282: main (in /home/sreekar/Documents/cpp/Valgrind)
==49694==
==49694== LEAK SUMMARY:
==49694==   definitely lost: 100 bytes in 2 blocks
==49694==   indirectly lost: 0 bytes in 0 blocks
==49694==   possibly lost: 0 bytes in 0 blocks
==49694==   still reachable: 0 bytes in 0 blocks
==49694==   suppressed: 0 bytes in 0 blocks
==49694==
==49694== For lists of detected and suppressed errors, rerun with: -s
==49694== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Figure 2: Case1: Commenting line 20

```
sreekar@sreekar-IdeaPad:~/Documents/cpp$ valgrind --leak-check=full ./Valgrind
==51791== Memcheck, a memory error detector
==51791== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==51791== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==51791== Command: ./Valgrind
==51791==
Dynamically allocated size in f1 = 60
arr[0] = 1
arr[1] = 3
arr[2] = 5
arr[3] = 7
arr[4] = 9
arr[5] = 11
arr[6] = 13
arr[7] = 15
arr[8] = 17
arr[9] = 19
arr[10] = 21
arr[11] = 23
arr[12] = 25
arr[13] = 27
arr[14] = 29
Dynamically allocated size in f2 = 40
==51791== Invalid write of size 4
==51791== at 0x109288: f2 (in /home/sreekar/Documents/cpp/Valgrind)
==51791== by 0x109283: main (in /home/sreekar/Documents/cpp/Valgrind)
==51791== Address 0x4a7f528 is 0 bytes after a block of size 40 alloc'd
==51791== at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-and64-linux.so)
==51791== by 0x109238: f2 (in /home/sreekar/Documents/cpp/Valgrind)
==51791== by 0x109283: main (in /home/sreekar/Documents/cpp/Valgrind)
==51791==
==51791== HEAP SUMMARY:
==51791==   in use at exit: 60 bytes in 1 blocks
==51791== total heap usage: 3 allocs, 2 frees, 1,124 bytes allocated
==51791==
==51791== 60 bytes in 1 blocks are definitely lost in loss record 1 of 1
==51791== at 0x4846828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-and64-linux.so)
==51791== by 0x1091AC: f1 (in /home/sreekar/Documents/cpp/Valgrind)
==51791== by 0x1092AE: main (in /home/sreekar/Documents/cpp/Valgrind)
==51791==
==51791== LEAK SUMMARY:
==51791==   definitely lost: 60 bytes in 1 blocks
==51791==   indirectly lost: 0 bytes in 0 blocks
==51791==   possibly lost: 0 bytes in 0 blocks
==51791==   still reachable: 0 bytes in 0 blocks
==51791==   suppressed: 0 bytes in 0 blocks
==51791==
==51791== For lists of detected and suppressed errors, rerun with: -s
==51791== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Figure 3: Case2: Keeping line 20

### Case2: keeping line 20

- The program had a total heap usage of (3 allocs, 2 frees, 1,124 bytes allocated), meaning that out of the three memory allocations.
- Two were freed, and one was not, leading to 60 bytes remaining allocated.
- Here f2 allocated memory is reclaimed, but not f1 allocated memory, which is  $4 \times 15 = 60$  Bytes.
- 'Address 0x4a7f528 is 0 bytes after a block of size 40 alloc'd:' says that function f2() attempts to write beyond the allocated memory block, causing an invalid write error.

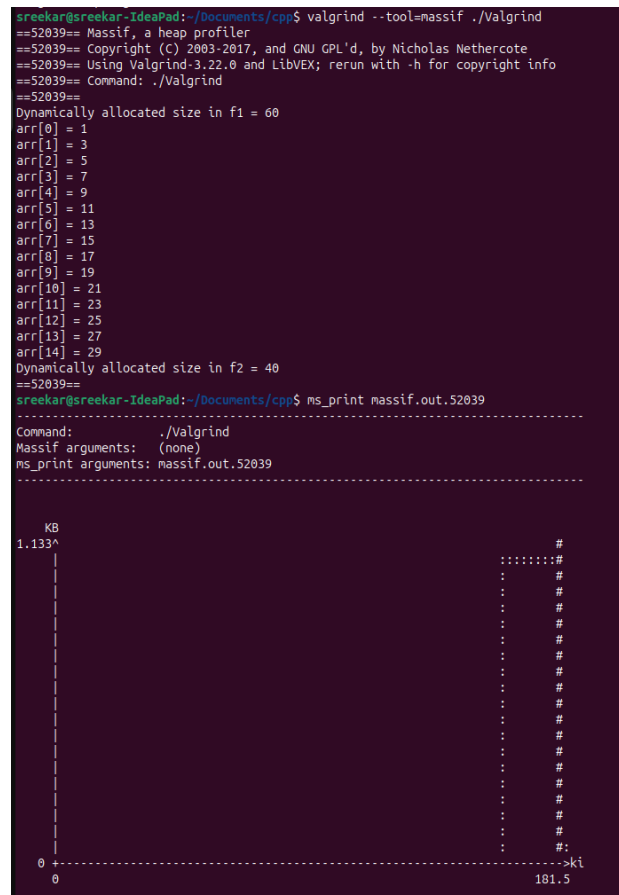
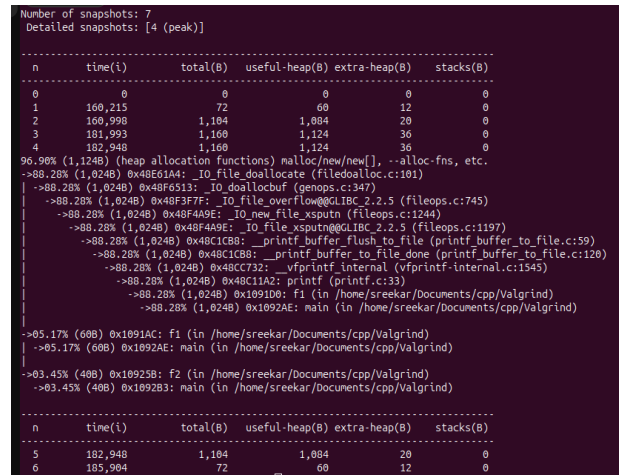


Figure 4:

## Massif tool

- Using this command 'valgrind --tool=massif file\_dir', all of Massif's profiling data is written to a file.
- By default, this file is called massif.out.pid, where pid is the process ID.
- Now run the command 'ms\_print massif.out.pid' which prints the detailed memory consumption, but only heap memory.(here pid = 52039)





```
Number of snapshots: 7
Detailed snapshots: [4 (peak)]

-----
n      time(t)      total(B)  useful-heap(B)  extra-heap(B)  stacks(B)
-----
0         0         0         0         0         0
1    160,215         72         60         12         0
2    160,998       1,104       1,084         20         0
3    181,993       1,160       1,124         36         0
4    182,948       1,160       1,124         36         0
96.90% (1,124B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->88.28% (1,024B) 0x48E61A4: _IO_file_doallocate (filedoalloc.c:101)
->88.28% (1,024B) 0x48F6513: _IO_dallochuf (genops.c:347)
->88.28% (1,024B) 0x48F37F7: _IO_file_overflow@@GLIBC_2.2.5 (fileops.c:745)
->88.28% (1,024B) 0x48F4A9E: _IO_new_file_xsputn (fileops.c:1244)
->88.28% (1,024B) 0x48F4A9E: _IO_file_xsputn@@GLIBC_2.2.5 (fileops.c:1197)
->88.28% (1,024B) 0x48C1C83: _printf_buffer_flush_to_file (printf_buffer_to_file.c:59)
->88.28% (1,024B) 0x48C1C80: _printf_buffer_to_file_done (printf_buffer_to_file.c:120)
->88.28% (1,024B) 0x48CC732: _vfprintf_internal (vfprintf-internal.c:1545)
->88.28% (1,024B) 0x48C11A2: printf (printf.c:33)
->88.28% (1,024B) 0x1091D0: f1 (in /home/sreekar/Documents/cpp/Valgrind)
->88.28% (1,024B) 0x1092AE: main (in /home/sreekar/Documents/cpp/Valgrind)

->05.17% (60B) 0x1091AC: f1 (in /home/sreekar/Documents/cpp/Valgrind)
->05.17% (60B) 0x1092AE: main (in /home/sreekar/Documents/cpp/Valgrind)

->03.45% (40B) 0x10925B: f2 (in /home/sreekar/Documents/cpp/Valgrind)
->03.45% (40B) 0x1092B3: main (in /home/sreekar/Documents/cpp/Valgrind)

-----
n      time(t)      total(B)  useful-heap(B)  extra-heap(B)  stacks(B)
-----
5    182,948       1,104       1,084         20         0
6    185,904         72         60         12         0
```

- Here 'total(B)' is the total number of bytes allocated on the heap at that time.
- 'useful-heap(B)' is the portion of the heap that was actively in use, 'extra-heap(B)' is extra heap memory allocated for reasons like alignment, padding, or fragmentation.
- 'stacks(B)' shows stack memory usage, it is 0 since massif focuses on heap memory.
- (96.90%)The majority of heap allocation is from functions like malloc, new, new[], and other allocation functions.
- 5.17% of the total memory (60 bytes) is allocated in the f1() function and later in the main() function.
- 3.45% of the total memory (40 bytes) is allocated in the f2() function and tracked during the program's execution.
- Overall, massif can help identify memory leaks by showing how heap memory is used throughout the program's execution. However, it is more indirect and requires careful analysis of the memory usage patterns.

### RAII (Resource Allocation is Initialization):

- Here resources are bounded by a scope at the time of initialization, if object out of scope destructor is called automatically
- Due to LIFO structure stack follows RAII by default but Heap memory needs to taken care of using smart pointers
- It is difficult to use RAII in case of Multi-threading environments, may end up with deadlocks.
- Also RAII is limited to C++, Rust and similar languages, not applicable Java and Python which doesn't have deterministic destructors.
- So, languages like C#, java and python use a built-in feature called GC(Garbage Collection), which is a form of automatic memory management

## 2 Garbage Collection

It is essential to first define what constitutes "garbage" in the context of memory management. Also the need and purpose of GC must be understood by now.

### 2.1 Cells and Liveness

- **Garbage:** Unused or unreachable objects in a program's memory. Thus they are no longer needed by the program and their memory should be reclaimed.
- **Cell:** A unit of heap memory, which holds data or a reference to another cell.
- **Root:** Pointers/references that are directly accessible by the program. They include local variables, global variables, stack, CPU registers ..etc.
- A cell is live if its address is held by any other root or a live cell.
- So simply cells which are not live are considered are garbage.

## 3 Garbage Collection Techniques

- 1) Reference Counting
- 2) Tracing (Mark and sweep, generational GC ..etc)
- 3) Modern GC techniques (G1, Parallel, CMS, ZGC ..etc)

### 3.1 Reference Counting

- The first GC technique is mark and sweep, which stores entire references graph. It marks live cells first and then sweeps garbage.
- It is computation intensive, have pauses and may lead to fragmentation of memory.
- Reference counting technique is later introduced, which is essentially keeping track of indegree of graph instead of storing it.
- Reference counting is very easy to understand and implement.

#### **Steps involved in Reference Counting Algorithm:**

**Initialization** - As a object is created, its reference count is set to be 1.

**Incrementing** - The reference count is increased if a new reference to object is made.

**Decrementing** - The reference count is decreased if a reference to object is removed.

**Deallocation** - If the reference count of that object drops to 0 then that memory is deallocated/reclaimed.

### 3.2 Disadvantages of Reference Counting

- Updating reference count every time a reference is altered, introduces runtime overhead.
- Reference counting and pointer load/store operations need to be atomic to ensure thread safety and avoid race conditions in concurrent environments.

**Algorithm 1** Reference Counting Pseudocode

---

```
procedure CREATE_OBJECT
  object  $\leftarrow$  allocate_memory()
  object.reference_count  $\leftarrow$  1
  return object
end procedure
procedure ADD_REFERENCE(object)
  if object is not null then
    object.reference_count  $\leftarrow$  object.reference_count + 1
  end if
end procedure
procedure REMOVE_REFERENCE(object)
  if object is not null then
    object.reference_count  $\leftarrow$  object.reference_count - 1
    if object.reference_count = 0 then
      deallocate_memory(object)
    end if
  end if
end procedure
procedure ASSIGN_REFERENCE(variable, new_object)
  old_object  $\leftarrow$  variable
  add_reference(new_object)
  variable  $\leftarrow$  new_object
  remove_reference(old_object)
end procedure
```

---

- Handling cyclic references is not possible in reference counting as their count never drops to zero and results in memory leaks.
- But if we use reference counting along with some periodic garbage collectors, this can be well suited for certain environments.

## References

- To know more about memory leak and Program execution [Memory\\_leaks](#)
- For further references about Valgrind see [Valgrind\\_documentation](#)
- More about smart pointer in [Smart\\_pointers](#)
- To know about cycle collection algorithms in reference counted systems, see [Concurrent Cycle Collection in Reference Counted Systems](#) Authors: David F. Bacon and V.T. Rajan, IBM T.J. Watson Research Center.