# Understanding Valgrind memory leak reports

Aleksander Morgado aleksander@es.gnu.org

*Thanks to the development team of Azetti Networks not only for supplying so many example memory leaks, but also for their useful comments in this document*

Document version 0.2 (February 4, 2010)

## 1 Introduction

The VALGRIND framework is a powerful tool to debug your applications, and specially for memory allocation related bugs. You can get a detailed explanation of all VALGRIND tools in the main site of the project (http://valgrind.org) and specially reading the VALGRIND User manual. VALGRIND can help to analyze any kind of software, in whatever language, including C, C++, Java, Fortran or assembly. Anyway, it is mostly used in C and C++ applications, mainly because the kind of errors found by Valgrind are usually more likely to happen in these two languages.

This article will show you the memory leak detector available in the MEMCHECK tool, which is just a slice of what VALGRIND really is. MEMCHECK is the default tool used when the `valgrind` command is executed without an explicit `--tool` option).

MEMCHECK tool acts at two different times: while program is being executed, and after program execution. As you may already know, memory leaks can only be detected **after** the program execution. This article will then only focus on that last report the `Valgrind` command generates after the process execution.

This article assumes that you already know and understand at least basically how the memory map in GNU/Linux system works, and specially the difference between memory statically allocated in the `stack` and memory dynamically allocated in the `heap`.

You can reproduce all types of memory leaks by compiling and using the simple tester available in http://es.gnu.org/∼aleksander/valgrind/valgrind-memcheck.c, which is released in the public domain. The tester is written in pure C, as this article is primarily focused on C programming.

## 1.1 Memory leaks

Blocks of statically allocated memory, those available in the `stack` of the process, will be available as long as the program runs, but only in the specific execution context of each moment. For example, variables declared at the beginning of a given function will only be available as long as the execution stays inside that specific function (or in a lower context). You can view as if those blocks of memory in `stack` are automatically deallocated for you during the execution of the program when moving to upper contexts..

On the other hand, you can dynamically allocate a block of memory in `heap` inside a given function, and return the pointer to that block of memory (address of the block) to the upper context, making it available outside of the context where it was originally allocated. The GNU C Library provides several methods to manage this memory in heap, but the most common ones are `malloc` and its variants[1] (in `malloc.h`), which provide an unconstrained way of allocation in heap:

```
void *malloc(size_t size);
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

In C, the programmer is in charge of explicitly deallocating that block of memory when no longer is going to be used. Other programming languages implement a mechanism to avoid this deallocation requirement, usually called a `garbage collector`, but we are going to avoid this as there are not many C programs using such implementation.

So, in a few words, **a memory leak occurs when a block of dynamically allocated memory** (using `malloc()` for example) **is never freed/deallocated explicitly** (with `free()`).

[1]Or any other method which uses `malloc()` internally, like `strdup()`

## 1.2 Memcheck

To avoid this kind of errors, VALGRIND provides the MEMCHECK memory checker, which will act primarily[2] on `heap` memory. As already stated, MEMCHECK will work at two different times; while process is being run, and after the process exits; but memory leaks as such can only be detected (by VALGRIND) once the program has finished.

The way MEMCHECK actually works could be subject of a long separate article, so you can just assume that MEMCHECK tracks all your calls to the dynamic memory allocator functions available in the `malloc` family[3]. Anyway, we will focus only on what MEMCHECK can find, and how it is reported in the output log of VALGRIND.

There are several options that can be passed to the `Valgrind` command when launching the MEMCHECK tool. The ones related to memory leaks, though, are not that many:

- `--leak-check=no|summary|full` : As MEMCHECK can be used for other purposes rather than memory leak checking, you have the possibility of disabling the leak check using the '`no`' value in this option, or just output a brief report of the memory leaks if using the '`summary`' value. In our case, '`full`' will be used for our tests, as we want to have not only a small summary, but also a detailed view of where the leaked memory was allocated (full backtrace for the exact point of allocation). Remember that the default value taken if not set in the command line is '`summary`'.

- `--leak-resolution=low|med|high` : MEMCHECK has three different 'resolution' levels, which only apply to the output report it generates, not to the way it will actually find the leaks. VALGRIND can 'merge' in a single report entry those leaks which are 'sufficiently similar'. To check for similarities, the backtrace of the allocation point is analyzed. In '`low`' level, only two entries in the backtrace need to match; four in '`med`' level; and all of them in '`high`' level (which by the way is also the default value if none explicitly set in the command line). We will use this default '`high`' leak resolution in our tests.

- `--show-reachable=no|yes` : MEMCHECK will not show neither 'Still Reachable' nor 'Indirectly Lost' leaks unless explicitly asked in the command line parameters. In our case, we also need to show those both type of leaks to be able to explain them, so '`yes`' should be used.

[2]VALGRIND may also report Invalid Read/Write warnings on variables in the `stack`, but not memory leaks
[3]For C++ developers, also the `new` and `delete` functions are tracked

You can get a whole list of options to be passed to the `valgrind` command, including those not related to the MEMCHECK tool, executing it with the `--help` option.

As already said, all tests shown in this document are executed using:

`valgrind --show-reachable=yes --leak-check=full <program> <arguments>`.

## 1.3 Pointers

In C programming language, a `pointer` is just a type of variable which can hold any memory address (either in `stack` or `heap`). The size of the pointer variable is given by the specific architecture of the Operating System, and defines the highest addressable memory. For example, in a 32 bit system, a pointer can hold any memory address up to $2^{32}$ (4 GBytes); and in a 64 bit system, any memory address up to $2^{64}$ (16 EBytes).

### 1.3.1 `start-pointers` and `interior-pointers`

There are two ways a given memory block can be reached. The first is using a `start-pointer`, which is a pointer which contains the start address of the block. The second way is using an `interior-pointer`, which is a pointer containing an address inside the allocated block.

Assuming that the `interior-pointer` is an expected situation, in theory the programmer could move along the `interior-pointer` until the beginning of the block, and thus forming a valid `start-pointer` that could be used to get the block deallocated.

The following figure shows an example of this kind of pointers, in a 5-byte allocated block of memory in `heap`.

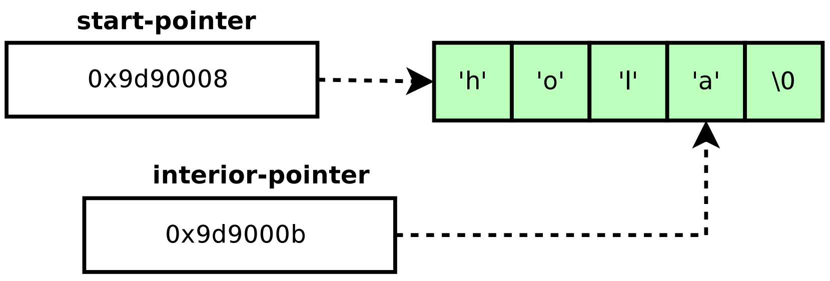

Figure 1: Start and Interior pointers

Please, note that `interior-pointers` must not be used in calls to `free()` [4]. In order to properly deallocate a block of memory which was dynamically allocated, only the `start-pointer` can be used.

One of the major limitations regarding the `interior-pointers` is that `Memcheck` cannot tell exactly their validity. This is, for `Memcheck`, there is no difference between these three cases (all are treated as `interior-pointers`):

[4]If you ever try to deallocate a block of memory using an `interior-pointer`, VALGRIND will report an 'Invalid free() / delete / delete[]' error.

- A start pointer which is re-used and moved along the block of memory
- A new pointer pointing to somewhere in the block of memory
- A random value in memory which accidentally happens to represent an address inside the block of memory

### 1.3.2 `chain of pointers`

A **chain** of pointers is created when the `start-pointer` or `interior-pointer` of a given memory block is only available in another different memory block. Imagine the situation where you allocate in heap a block of memory (call it AAA) to hold a `struct`, and inside the `struct` you have a pointer which you then use to store the address of a newly allocated block of memory (call it BBB). In this case, the `start-pointer` of BBB is stored in AAA, so you end up with a `chain of start-pointers` as BBB is reachable through AAA.

The easiest way of visualizing such a chain of pointers is thinking in a standard linked-list in C, where each element of the list contains a pointer to the next element, and the list itself is fully referenced by an initial 'head' pointer (`start-pointer` of the first element in the list).

## 1.4 Brief description of the different possible memory leaks

The Valgrind User Manual includes the following list of 9 memory leak types. Any possible kind of memory leak should fall into one of these categories.

```
     Pointer chain            AAA Category    BBB Category
     -------------            ------------    ------------
(1)  RRR ------------> BBB                    DR
(2)  RRR ---> AAA ---> BBB    DR              IR
(3)  RRR               BBB                    DL
(4)  RRR      AAA ---> BBB    DL              IL
(5)  RRR ------?-----> BBB                    (y)DR, (n)DL
(6)  RRR ---> AAA -?-> BBB    DR              (y)IR, (n)DL
(7)  RRR -?-> AAA ---> BBB    (y)DR, (n)DL    (y)IR, (n)IL
(8)  RRR -?-> AAA -?-> BBB    (y)DR, (n)DL    (y,y)IR, (n,y)IL, (_,n)DL
(9)  RRR      AAA -?-> BBB    DL              (y)IL, (n)DL

  Pointer chain legend:
  - RRR: a root set node or DR block
  - AAA, BBB: heap blocks
  - --->: a start-pointer
  - -?->: an interior-pointer

  Category legend:
  - DR: Directly reachable
  - IR: Indirectly reachable
  - DL: Directly lost
  - IL: Indirectly lost
  - (y)XY: it's XY if the interior-pointer is a real pointer
  - (n)XY: it's XY if the interior-pointer is not a real pointer
  - (_)XY: it's XY in either case
```

The previous list of leak types will be deeply analyzed in the following sections, so it's OK if they are not fully understood yet.

Anyway, even if internally Valgrind distinguishes 9 different types of memory leaks, the generated output report will only include 4 main categories:

| | |
|---|---|
| **Still Reachable:** | Covers cases 1 and 2 (for the BBB blocks) |
| **Directly Lost:** | Covers case 3 (for the BBB blocks) |
| **Indirectly Lost:** | Covers cases 4 and 9 (for the BBB blocks) |
| **Possibly Lost:** | Covers cases 5, 6, 7 and 8 (for the BBB blocks) |

Directly and Indirectly Lost leaks are also referred as **Definitely Lost** leaks.

### 1.4.1 Still reachable blocks

A block of memory is reported to be **Still Reachable** when MEMCHECK finds, after process execution ends, at least one pointer with the start address of the block (a `start-pointer`).

This pointer found can be either in stack (a global pointer, or a pointer in the `main` function), or in heap if it was referenced by another `start-pointer` (which again could be in the stack, or referenced by another `start-pointer`, thus forming a chain of `start-pointers`).

Usually, 'Still Reachable' memory leaks are not considered harmful in the standard development. There are stable and popular libraries in the Free Software world (talking about `GLib` and `GTK+` here) which dynamically allocate a lot of heap memory for data which should be available during the whole execution of the program; this is, memory which shouldn't be deallocated until program is about to exit. And in this situation, instead of explicitly freeing this memory before the return of the main function, it's just easier and faster to leave the kernel do it automatically when the process ends.

### 1.4.2 Definitely Lost blocks

A leak is considered **Definitely Lost** when, at process exit, there is no pointer or chain of pointers to the leaked memory block:

- When the `start-pointer` of a block of memory is fully lost, and also no other `interior-pointer` to that block of memory is available when process ends; MEMCHECK will report that block of memory as a '**Directly Lost**' leak.
- When MEMCHECK finds a valid `start-pointer` or `interior-pointer` to a given block of memory, but that pointer is in another block which is 'Directly Lost', MEMCHECK will report the block of memory as an '**Indirectly Lost**' leak.
- Finally, when MEMCHECK finds a valid `start-pointer` or `interior-pointer` to a given block of memory, but that pointer is in another block which is 'Indirectly Lost', MEMCHECK will report the block of memory as also being 'Indirectly Lost'.

The 'Definitely Lost' memory leaks (both Direct and Indirect) are the ones the programmer should pay more attention to, as they clearly show **real programming errors**.

### 1.4.3 Possibly Lost blocks

The last type of leaks reported by Memcheck are the **Possibly Lost** ones, which also are usually the most difficult to understand.

The term 'possibly' here states that Valgrind does not know whether the leak is 'Definitely Lost' or 'Still Reachable': but the leaks are really of one of these two types. In other words, it's a task for the programmer to check whether the leak is reachable or not.

The 'Possibly Lost' leaks will be reported in the absence of a valid `start-pointer` to the block, but when at least one `interior-pointer` is found. As you already know, Memcheck cannot decide if the `interior-pointer` is a valid one, or just a funny coincidence. As a rule of thumb, anyway, those `interior-pointers` should be treated as valid ones (and decide the real type of leak based on that).

## 1.5 The output memory leak report

Valgrind can output the memory leak report in several formats, including XML, but this document will focus on the best human-readable way, which is the standard text report.

The following is a simple output report example:

```
HEAP SUMMARY:
    in use at exit: 4 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 4 bytes allocated

4 bytes in 1 blocks are still reachable in loss record 1 of 1
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x40B0CDF: strdup (strdup.c:43)
   by 0x804879B: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 4 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
```

The report is divided into two main parts:

- **Heap Summary:** This report will show some statistical values of the execution, and what is more important, the whole list of allocation backtraces[5].
  - For each memory leak Memcheck finds, an allocation backtrace will be shown. The allocation backtrace shows the whole stack of functions indicating where the memory block was allocated.
  - If the same memory leak is found several times (this is, a leak of same type with same backtrace[6]), the output report will show a combined single backtrace for all. If several leaks are grouped due to this reason, the report will show the number of blocks leaked due to this reason, and the sum of the size of all blocks.
  - If the executed program was compiled with debugging symbols and was not stripped, or if the stripped info is available in a different file (like with standard 'debug' packages in RPM in Deb based systems), the report will show the function names, the source file name and the exact line in the source file where the allocation was done.
  - Another special point in the allocation backtrace is that for 'Directly Lost' leaks which

[5] Only if `--leak-check=full` input option is specified
[6] Note that this behavior depends on the value of the `--leak-resolution` input option

generate at the same time some 'Indirectly Lost' ones, the report will group together both in the leak report: the directly leaked ones and the indirectly leaked ones. The report will specify how many of each type, and the total amount of leaked bytes. Note anyway, that an independent report line will also be included for the Indirectly Lost ones, showing its own backtrace. We will see some examples of this situation in the analyzed cases.

- **Leak Summary:** Finally, a short summary is included in the output report, which will usually tell you in a fast way if your application is a leak-killer or a leak-generator.
  - One of the important things to consider here is that the naming of each item in the summary is not very accurate. The term '`definitely lost`' in the output Leak Summary refers only to the 'Directly Lost' leaks, without considering the 'Indirectly Lost' ones (which of course are also 'Definitely Lost', but listed separately).

# 2 Leaks tracking `start-pointers`

## 2.1 Case 1. Directly Reachable in BBB

The first kind of memory leak is probably the easiest one to follow. It implies having a `RRR` pointer which is available at program exit (so for example a global variable) and a single memory allocation.

This type of memory leak is referenced as Case 1 in the Valgrind User Manual:

```
     Pointer chain            AAA Category    BBB Category
     -------------            ------------    ------------
(1)  RRR ------------> BBB                    DR
```

As you may have already noticed, this kind of memory leak doesn't imply having an additional AAA intermediate block of memory.

The following lines of code show how to reproduce this case.

```
void *rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated BBB block */
    rrr = strdup("bbb");

    return 0;
}
```

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

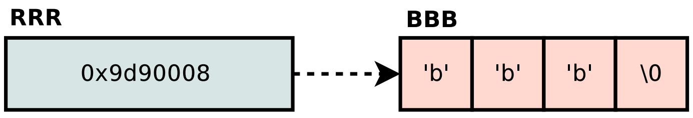

Figure 2: Directly Reachable memory leak in BBB (Case 1)

In this case, when program exits, the RRR pointer is a valid `start-pointer` pointing to the beginning of the BBB memory block (address `0x9d90008`), and this memory leak will be internally detected as a **Directly Reachable** memory leak, which will then be included in the list of 'Still Reachable' leaks in the output report.

The output that VALGRIND generates for the previous situation is as follows:

```
HEAP SUMMARY:
    in use at exit: 4 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 4 bytes allocated

4 bytes in 1 blocks are still reachable in loss record 1 of 1
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x40B0CDF: strdup (strdup.c:43)
   by 0x804879B: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 4 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
```

If you read it carefully, you will see that there is no reference to 'Directly Reachable' memory, as the leak is treated as a more global 'Still Reachable'. Some other things to consider in the output report are:

- **Allocation backtrace:** VALGRIND shows the full backtrace for the allocation of the BBB block, and includes the size of the leak (4bytes) and the amount of blocks leaked (1 block).
- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, only 1 block of 4 bytes was leaked, and is 'Still Reachable'.

## 2.2 Case 2. Directly Reachable in AAA, Indirectly Reachable in BBB

This second kind of memory leak is fully related to the previous one. In this situation, RRR will point to an intermediate memory block (AAA) which holds another pointer to another independent memory block (BBB), thus having a chain of 2 `start-pointers`.

This type of memory leak is referenced as Case 2 in the Valgrind User Manual:

```
      Pointer chain            AAA Category    BBB Category
      -------------            ------------    ------------
 (2)  RRR ---> AAA ---> BBB    DR              IR
```

The following lines of code show how to reproduce this case.

```
void **rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated AAA block */
    rrr = malloc(sizeof(void **));

    /* Store in AAA the address of the newly allocated BBB block */
    *rrr = strdup("bbb");

    return 0;
}
```

In a brief, the address of a newly allocated block (AAA) is stored in RRR. The AAA block has enough size to hold a pointer, and in this block the address of another newly allocated block (BBB) is then stored.

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

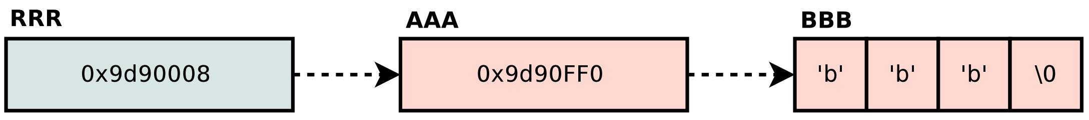

Figure 3: Indirectly Reachable memory leak in BBB (Case 2)

When program ends, MEMCHECK will find referenced by RRR a Directly Reachable block (AAA), and in the same chain of `start-pointers`, will also find the BBB block, which will be treated as **Indirectly Reachable**.

The output that VALGRIND generates for the previous situation is as follows:

```
HEAP SUMMARY:
    in use at exit: 8 bytes in 2 blocks
  total heap usage: 2 allocs, 0 frees, 8 bytes allocated

4 bytes in 1 blocks are still reachable in loss record 1 of 2
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x80487A2: main (in /home/aleksander/valgrind-memcheck)

4 bytes in 1 blocks are still reachable in loss record 2 of 2
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x40B0CDF: strdup (strdup.c:43)
   by 0x80487A2: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 8 bytes in 2 blocks
        suppressed: 0 bytes in 0 blocks
```

In detail:

- **Allocation backtrace:** VALGRIND shows one record as a 'Directly Reachable' leak (AAA block) and the other one is 'Indirectly Reachable' one (BBB) block. As in Case 1, the output report doesn't specify this exact differences detected internally, and just reports that both are 'Still Reachable'.
- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, 2 blocks with a total size of 8 bytes were leaked and 'Still Reachable'.

## 2.3 Case 3. Directly Lost in BBB

This type of memory leak is referenced as Case 3 in the Valgrind User Manual:

```
     Pointer chain            AAA Category    BBB Category
     -------------            ------------    ------------
(3)  RRR            BBB                       DL
```

In order to explain this case, the source code for Case 1 is used, just adding one more line simulating the effect of losing the start address of the block by resetting to `NULL` the contents of RRR. Remember that RRR is a valid and reachable pointer when program ends.

```
void *rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated BBB block */
    rrr = strdup("bbb");

    /* oops, we lose the start address of the BBB block */
    rrr = NULL;

    return 0;
}
```

When allocating the BBB block, the start address was stored in the RRR pointer, but before the program ends, the address is lost as RRR is reseted to `NULL`.

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

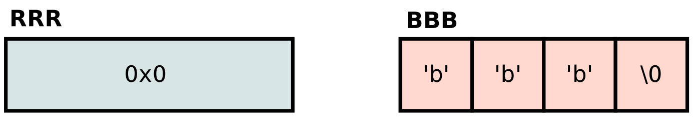

Figure 4: Directly Lost in BBB (Case 3)

VALGRIND could not find a single pointer (neither `start-pointer` nor `interior-pointer`) to the BBB block, as RRR (the only one that was ever available in the program) is `NULL`. This type of memory leak is named as **Directly Lost**.

The output that VALGRIND generates for the previous situation is as follows:

```
HEAP SUMMARY:
     in use at exit: 4 bytes in 1 blocks
   total heap usage: 1 allocs, 0 frees, 4 bytes allocated

 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
    by 0x40B1CDF: strdup (strdup.c:43)
    by 0x80487A9: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
    definitely lost: 4 bytes in 1 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 0 bytes in 0 blocks
    still reachable: 0 bytes in 0 blocks
         suppressed: 0 bytes in 0 blocks
```

The output report of VALGRIND seems not very clear regarding the difference between 'Directly Lost' and 'Definitely Lost':

- **Allocation backtrace:** VALGRIND shows the full backtrace for the allocation of the BBB block, and includes the size of the leak (4bytes) and the amount of blocks leaked (1 block). In the allocation backtrace report, the leak is identified as 'Definitely Lost', but the programmer must take into account that the leak is 'Directly Lost'.
- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, only 1 block of 4 bytes was leaked, and is shown as 'Definitely Lost'. Again, the programmer should read 'Directly Lost' here.

All the 'Directly Lost' memory blocks are reported as **Definitely Lost**.

## 2.4 Case 4. Directly Lost in AAA, Indirectly Lost in BBB

This type of memory leak is referenced as Case 4 in the Valgrind User Manual:

```
       Pointer chain               AAA Category     BBB Category
       -------------               ------------     ------------
 (4)   RRR        AAA ---> BBB     DL               IL
```

In order to explain this case, the source code of Case 2 will be used, just adding one more line, simulating the effect of losing the start address of the block by resetting to `NULL` the contents of RRR.

```
void **rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated AAA block */
    rrr = malloc(sizeof(void **));


    /* Store in AAA the address of the newly allocated BBB block */
    *rrr = strdup("bbb");

    /* oops, we lose the start address of the AAA block */
    rrr = NULL;

    return 0;
}
```

So again, in RRR pointer the address of a newly allocated memory block (AAA) is stored, which has enough space to store a pointer, which is filled in with the address of another newly allocated memory block (BBB). Then, the RRR pointer is reseted to `NULL` and both blocks are leaked.

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

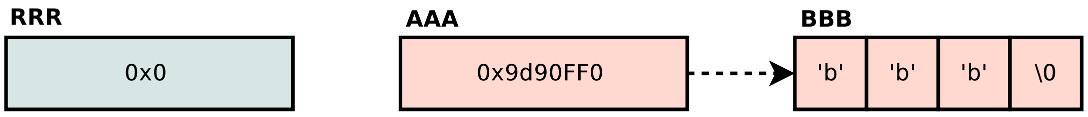

Figure 5: Indirectly Lost in BBB (Case 4)

VALGRIND could not find a single pointer to the AAA block, as RRR is `NULL`, so AAA is marked as Directly Lost. But in this case, AAA still has the address of the BBB block (AAA holds the `start-pointer` of BBB), so VALGRIND will report BBB as a **Indirectly Lost** leak.

The output report of VALGRIND for this case will show:

```
HEAP SUMMARY:
    in use at exit: 8 bytes in 2 blocks
  total heap usage: 2 allocs, 0 frees, 8 bytes allocated

4 bytes in 1 blocks are indirectly lost in loss record 1 of 2
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x40B1CDF: strdup (strdup.c:43)
   by 0x8048774: main (in /home/aleksander/valgrind-memcheck)

8 (4 direct, 4 indirect) bytes in 1 blocks are definitely lost in
                           loss record 2 of 2
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x8048774: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
   definitely lost: 4 bytes in 1 blocks
   indirectly lost: 4 bytes in 1 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

The output report of VALGRIND seems again not very clear regarding the difference between 'Directly Lost', 'Indirectly Lost' and 'Definitely Lost':

- **Allocation backtrace:**
  - VALGRIND shows the full backtrace for the allocation of the BBB block, and includes the size of the leak (4bytes) and the amount of blocks leaked (1 block). BBB block backtrace reports the leak as 'Indirectly Lost'. Note that this allocation backtrace will only be shown if the `--show-reachable=yes` option is used when calling the `valgrind` command.
  - The allocation backtrace of AAA is a little bit more complicated. It will say that 1 block is leaked (AAA), but the total size of leaked bytes considers both the AAA and BBB blocks (8 bytes), as they were in a chain of pointers. In this specific case, the output report is correct saying that 8 bytes are 'Definitely Lost', but the programmer should understand that 4 of those bytes were 'Indirectly Lost' (as specified in the allocation backtrace of BBB) and 4 bytes 'Directly Lost' (this AAA backtrace).
- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, only 1 block of 4 bytes is shown as 'Definitely Lost' (AAA), but the truth is that both BBB and AAA are 'Definitely Lost'. BBB is shown apart as 'Indirectly Lost'. In other words, the line '`definitely lost`' here should be treated specifically as 'Directly Lost'.

# 3 Leaks tracking `interior-pointers`

## 3.1 Case 5. Possibly Lost in BBB, with an interior-pointer to BBB

The simplest case using `interior-pointers` is a modification of the previous Case 3, where the RRR pointer is setup as an `interior-pointer` holding an address which points to somewhere inside the block BBB.

This type of memory leak is referenced as Case 3 in the Valgrind User Manual:

```
     Pointer chain             AAA Category    BBB Category
     -------------------       ------------    ------------
(5)  RRR ------?-----> BBB                     (y)DR, (n)DL
```

As seen in the diagram above, Memcheck only finds in RRR a pointer containing an address inside the BBB block (an `interior-pointer`).

Depending on the validity of the `interior-pointer`, which must be decided by the programmer, the leak will be:

- **(y)DR:** 'Directly Reachable' if the `interior-pointer` is a valid one.
- **(n)DL:** 'Directly Lost' if the `interior-pointer` is not valid.

The following code shows this situation:

```c
void *rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated BBB block */
    rrr = strdup("bbb");

    /* We setup a valid interior-pointer to 1 byte inside BBB
     * (start-pointer to BBB is lost)
     */
    rrr = ((char *)rrr)+1;

    return 0;
}
```

Once program ends, Memcheck sees that there is no `start-pointer` for BBB, but it can find a possible valid `interior-pointer` to BBB in RRR. Of course, in this case, it should be clear to the programmer that the leak is really 'Directly Reachable', as actually the `interior-pointer` was forced on purpose.

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

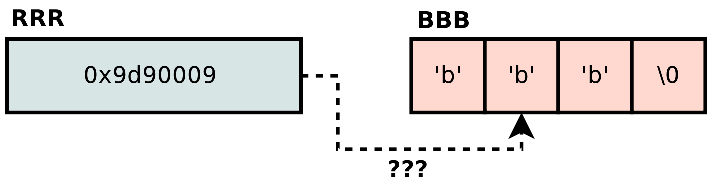

Figure 6: Possibly Lost in BBB (Case 5)

The output of VALGRIND will show the following:

```
HEAP SUMMARY:
    in use at exit: 4 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 4 bytes allocated

4 bytes in 1 blocks are possibly lost in loss record 1 of 1
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x40B1CDF: strdup (strdup.c:43)
   by 0x80487B7: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 4 bytes in 1 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

In detail,

- **Allocation backtrace:** VALGRIND shows the full backtrace for the allocation of the BBB block, and includes the size of the leak (4bytes) and the amount of blocks leaked (1 block). BBB block backtrace reports the leak as 'Possibly Lost'.
- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, only 1 block of 4 bytes is shown as 'Possibly Lost'.

## 3.2 Case 6. Directly Reachable in AAA, Possibly Lost in BBB, with an interior-pointer to BBB

This type of memory leak is referenced as Case 6 in the Valgrind User Manual:

```
     Pointer chain              AAA Category     BBB Category
     -------------------------  -------------    -------------
(6)  RRR ----> AAA -?-> BBB     DR               (y)IR, (n)DL
```

In this case, RRR holds the `start-pointer` of AAA (thus, AAA is a 'Directly Reachable' leak), which holds an `interior-pointer` to BBB.

Depending on the validity of the `interior-pointer`, which must be decided by the programmer, the leak will be:

- **(y)IR:** 'Indirectly Reachable' if the `interior-pointer` is a valid one.
- **(n)DL:** 'Directly Lost' if the `interior-pointer` is not valid.

An example of this case is:

```
void **rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated AAA block */
    rrr = malloc(sizeof(void **));

    /* Store in AAA the address of the newly allocated BBB block */
    *rrr = strdup("bbb");

    /* We setup a valid interior-pointer to 1 byte inside BBB
     * (start-pointer to BBB is lost)
     */
    *rrr = ((char *)(*rrr))+1;

    return 0;
}
```

Note that in the example above, AAA is forced to contain an `interior-pointer` to BBB, by applying an offset of 1 single byte to the start address of the block. Once program ends, Memcheck sees that there is no `start-pointer` for BBB, but it can find a possible valid `interior-pointer` to BBB in AAA, as AAA is 'Still Reachable'. Of course, in this case, it should be clear to the programmer that the leak is really 'Indirectly Reachable', as actually the `interior-pointer` was forced on purpose.

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

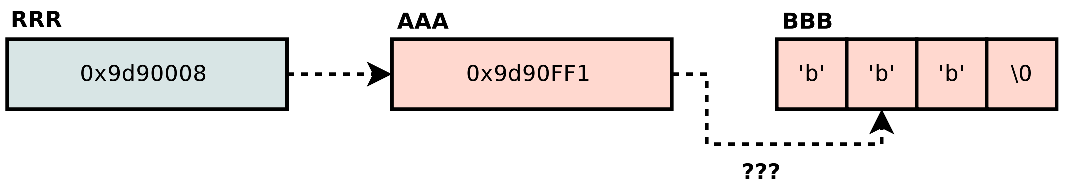

Figure 7: Possibly Lost in BBB (Case 6)

The output of VALGRIND will show the following:

```
HEAP SUMMARY:
     in use at exit: 8 bytes in 2 blocks
   total heap usage: 2 allocs, 0 frees, 8 bytes allocated

4 bytes in 1 blocks are still reachable in loss record 1 of 2
    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
    by 0x80487BE: main (in /home/aleksander/valgrind-memcheck)

4 bytes in 1 blocks are possibly lost in loss record 2 of 2
    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
    by 0x40B1CDF: strdup (strdup.c:43)
    by 0x80487BE: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
    definitely lost: 0 bytes in 0 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 4 bytes in 1 blocks
    still reachable: 4 bytes in 1 blocks
         suppressed: 0 bytes in 0 blocks
```

In detail,

- **Allocation backtrace:**

  - First, VALGRIND shows the full backtrace for the allocation of the AAA block, and includes the size of the leak (4bytes) and the amount of blocks leaked (1 block). AAA block backtrace reports the leak as 'Still Reachable'.

  - Second, the backtrace for the allocation of the BBB block is shown (4 bytes in 1 block). BBB is reported as 'Possibly Lost', because only an `interior-pointer` to BBB was found.

- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, 1 block of 4 bytes is shown as 'Possibly Lost', and 1 block of 4 bytes is shown as 'Still Reachable'.

As you may have already noticed, VALGRIND will not link in any way both leaks, even if there is a clear relation between them (AAA contains a pointer to BBB). Thus, tracking this kind of 'Possibly Lost' leaks becomes more difficult, and the programmer should then rely only on the specific source code reported in the backtraces.

## 3.3 Case 7. Possibly Lost in AAA and BBB, with an interior-pointer to AAA

This type of memory leak is referenced as Case 7 in the Valgrind User Manual:

```
      Pointer chain               AAA Category     BBB Category
      -------------------------   ------------     ------------
(7)   RRR -?-> AAA ---> BBB       (y)DR, (n)DL     (y)IR, (n)IL
```

When a leak is reported as 'Possibly Lost' (AAA), and the leaked block contains `start-pointers` to other blocks of memory (BBB), those are also reported as 'Possibly Lost'. Once the programmer decides if the `interior-pointer` of AAA is valid or invalid, it will not only know the type of leak for AAA, but also the type of leak for BBB.

Then depending on the validity of the `interior-pointer` stored in RRR, the leaks will be:

- **AAA (y)DR:** 'Directly Reachable' if the `interior-pointer` to AAA is a valid one.
- **AAA (n)DL:** 'Directly Lost' if the `interior-pointer` to AAA is not valid.
- **BBB (y)IR:** 'Indirectly Reachable' if the `interior-pointer` to AAA is a valid one.
- **BBB (n)IL:** 'Indirectly Lost' if the `interior-pointer` to AAA is not valid.

An example of this kind of leaks is as follows:

```c
void **rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated AAA block */
    rrr = malloc(sizeof(void **));

    /* Store in AAA the address of the newly allocated BBB block */
    *rrr = strdup("bbb");

    /* We setup a valid interior pointer to 1 byte inside AAA
     * (start-pointer to AAA is lost) */
    rrr = (void **)((char *)(rrr)+1);

    return 0;
}
```

Once program execution ends, MEMCHECK sees that there is no `start-pointer` for AAA, but it can find a possible valid `interior-pointer` to AAA in RRR. Also, in this case, it should be clear to the programmer that the AAA leak is really 'Directly Reachable' and the BBB leak 'Indirectly Reachable', as actually the `interior-pointer` to AAA was forced on purpose.

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

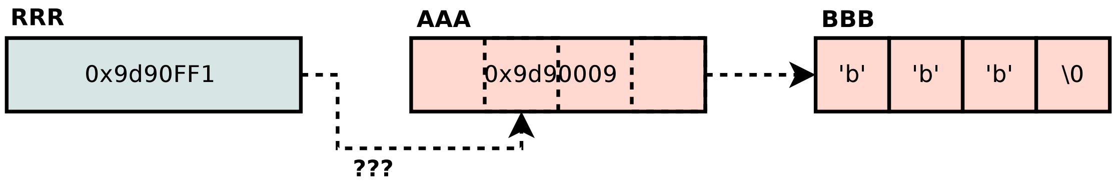

Figure 8: Possibly Lost in BBB and AAA (Case 7)

The output of VALGRIND will show the following:

```
HEAP SUMMARY:
    in use at exit: 8 bytes in 2 blocks
  total heap usage: 2 allocs, 0 frees, 8 bytes allocated

4 bytes in 1 blocks are possibly lost in loss record 1 of 2
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x80487C5: main (in /home/aleksander/valgrind-memcheck)

4 bytes in 1 blocks are possibly lost in loss record 2 of 2
   at 0x4024C1C: malloc (vg_replace_malloc.c:195)
   by 0x40B1CDF: strdup (strdup.c:43)
   by 0x80487C5: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 8 bytes in 2 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
```

In detail,

- **Allocation backtrace:**

  - First backtrace shows the report of the AAA allocation, as a 'Possibly Lost' leak in 1 block of 4 bytes.

  - Second backtrace shows the report of the BBB allocation, also as a 'Possibly Lost' leak in 1 block of 4 bytes.

- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, both 2 blocks with a total of 8 bytes is shown as 'Possibly Lost'.

As in the previous case, VALGRIND will not show any link between both leaks, even if there is a real linkage (AAA holds the `start-pointer` of BBB.

## 3.4 Case 8. Possibly Lost in AAA and BBB, with interior-pointers to both AAA and BBB

This type of memory leak is referenced as Case 8 in the Valgrind User Manual:

```
     Pointer chain              AAA Category   BBB Category
     ------------------------   ------------   ------------
(8)  RRR -?-> AAA -?-> BBB      (y)DR, (n)DL   (y,y)IR, (n,y)IL, (_,n)DL
```

The last case involving 'Possibly Lost' leaks happens when instead of a single `interior-pointer`, two are found by Memcheck: first one in RRR pointing to somewhere in the AAA block, and second one in AAA, pointing to somewhere in BBB. The programmer needs to decide now the validity of both pointers:

- **AAA (y)DR:** 'Directly Reachable' if the `interior-pointer` to AAA is a valid one.
- **AAA (n)DL:** 'Directly Lost' if the `interior-pointer` to AAA is not valid.
- **BBB (y,y)IR:** 'Indirectly Reachable' if the `interior-pointer` to AAA is a valid one, and if the `interior-pointer` to BBB is also valid.
- **BBB (n,y)IL:** 'Indirectly Lost' if the `interior-pointer` to AAA is valid, but the `interior-pointer` to BBB is not valid.
- **BBB (_,n)DL:** 'Directly Lost' if the `interior-pointer` to BBB is not valid, not depending on the validity of the pointer to AAA.

Graphically, this is what Memcheck sees when program ends (red squares show leaked block):

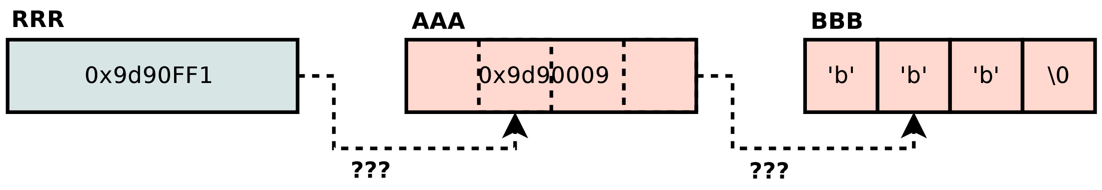

Figure 9: Possibly Lost in AAA and BBB (Case 8)

An example of these leaks is as follows:

```
void **rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated AAA block */
    rrr = malloc(sizeof(void **));

    /* Store in AAA the address of the newly allocated BBB block */
    *rrr = strdup("bbb");

    /* We setup in AAA, a valid interior pointer to 1 byte inside
     *  BBB (start-pointer to BBB is lost) */
    *rrr = ((char *)(*rrr)+1);

    /* We setup in RRR, a valid interior pointer to 1 byte inside
     *  AAA (start-pointer to AAA is lost) */
    rrr = (void **)(((char *)(rrr))+1);

    return 0;
}
```

Once process finishes, Memcheck finds both linked `interior-pointers`, and will mark both of them as 'Possibly Lost' ones. In this specific example, both pointers are really valid ones as we forced them, so the programmer should conclude that AAA is 'Directly Reachable' and BBB 'Indirectly Reachable' (as in Case 2).

The output of Valgrind will show the following:

```
HEAP SUMMARY:
     in use at exit: 8 bytes in 2 blocks
   total heap usage: 2 allocs, 0 frees, 8 bytes allocated

4 bytes in 1 blocks are possibly lost in loss record 1 of 2
    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
    by 0x80487CC: main (in /home/aleksander/valgrind-memcheck)

4 bytes in 1 blocks are possibly lost in loss record 2 of 2
    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
    by 0x40B1CDF: strdup (strdup.c:43)
    by 0x80487CC: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
    definitely lost: 0 bytes in 0 blocks
    indirectly lost: 0 bytes in 0 blocks
      possibly lost: 8 bytes in 2 blocks
    still reachable: 0 bytes in 0 blocks
         suppressed: 0 bytes in 0 blocks
```

In detail,

- **Allocation backtrace:**
  - First backtrace shows the report of the AAA allocation, as a 'Possibly Lost' leak in 1 block of 4 bytes.
  - Second backtrace shows the report of the BBB allocation, also as a 'Possibly Lost' leak in 1 block of 4 bytes.
- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, both 2 blocks with a total of 8 bytes is shown as 'Possibly Lost'.

If you compare this output report with the one reported in the previous Case 7, you will find them completely equal.

## 3.5 Case 9. Directly Lost in AAA, Definitely Lost in BBB, with an interior-pointer to BBB

This type of memory leak is referenced as Case 9 in the Valgrind User Manual:

```
     Pointer chain            AAA Category    BBB Category
     -------------            ------------    ------------
(9)  RRR      AAA -?-> BBB    DL              (y)IL, (n)DL
```

Last, but not least, Case 9 shows the only leak which is not 'Possibly Lost' when having `interior-pointers`. In this case, the intermediate AAA block is 'Directly Lost', and thus, BBB will be also 'Definitely Lost' (either Directly or Indirectly), depending on the validity of the `interior-pointer` to BBB stored in AAA:

- **BBB (y)IL:** 'Indirectly Lost' if the `interior-pointer` to BBB is a valid one.
- **BBB (n)DL:** 'Directly Lost' if the `interior-pointer` to BBB is not valid.

In a brief, BBB is always lost in this case, as AAA is also lost.

An example of this case is as follows:

```
void **rrr;

int main(void)
{
    /* Store in RRR the address of the newly allocated AAA block */
    rrr = malloc(sizeof(void **));

    /* Store in AAA the address of the newly allocated BBB block */
    *rrr = strdup("bbb");

    /* We setup in AAA, a valid interior pointer to 1 byte inside
     * BBB (start-pointer to BBB is lost) */
    *rrr = ((char *)(*rrr)+1);

    /* oops, we lose the start address of the AAA block */
    rrr = NULL;

    return 0;
}
```

Graphically, this is what MEMCHECK sees when program ends (red squares show leaked block):

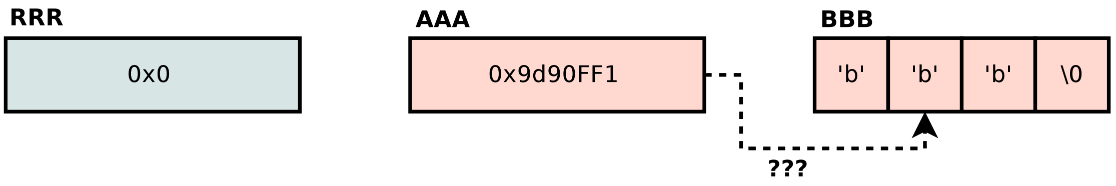

Figure 10: Possibly Lost in BBB (Case 9)

VALGRIND output will say:

```
HEAP SUMMARY:
     in use at exit: 8 bytes in 2 blocks
   total heap usage: 2 allocs, 0 frees, 8 bytes allocated

 4 bytes in 1 blocks are indirectly lost in loss record 1 of 2
    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
    by 0x40B1CDF: strdup (strdup.c:43)
    by 0x8048774: main (in /home/aleksander/valgrind-memcheck)

 8 (4 direct, 4 indirect) bytes in 1 blocks are definitely lost in
                           loss record 2 of 2
    at 0x4024C1C: malloc (vg_replace_malloc.c:195)
    by 0x8048774: main (in /home/aleksander/valgrind-memcheck)

LEAK SUMMARY:
    definitely lost: 4 bytes in 1 blocks
    indirectly lost: 4 bytes in 1 blocks
      possibly lost: 0 bytes in 0 blocks
    still reachable: 0 bytes in 0 blocks
         suppressed: 0 bytes in 0 blocks
```

In detail:

- **Allocation backtrace:**

  - VALGRIND shows the full backtrace for the allocation of the BBB block, and includes the size of the leak (4bytes) and the amount of blocks leaked (1 block). BBB block backtrace reports the leak as 'Indirectly Lost' (even if it could be 'Directly Lost' if the `interior-pointer` was not valid). Note that this allocation backtrace will only be shown if the `--show-reachable=yes` option is used when calling the `valgrind` command.

  - The allocation backtrace of AAA is a little bit more complicated. It will say that 1 block is leaked (AAA), but the total size of leaked bytes considers both the AAA and BBB blocks (8 bytes), as they were in a chain of pointers. In this specific case, the output report is correct saying that 8 bytes are 'Definitely Lost', but the programmer should understand that

4 of those bytes were 'Directly Lost' (AAA) and the other 4 bytes (BBB) either 'Directly Lost' or 'Indirectly Lost' (depending on the validity of the `interior-pointer` to BBB.

- **Leak Summary:** The leak summary shows the summary of all leak types. In this case, only 1 block of 4 bytes is shown as 'Definitely Lost' (AAA), but the truth is that both BBB and AAA are 'Definitely Lost'. BBB is shown apart as 'Indirectly Lost', even if it could also be 'Directly Lost'.

Again, if you compare this output report with the one in Case 4, you will find them completely equal.

## 4 Conclusions

Some key points to summarize.

- Usually, if you don't run `Valgrind` on your program frequently, you may end up with a long list of leak reports. The suggested order to fix them is the following one:
  - **Directly Lost** leaks (reported as 'Definitely Lost' in the output report)
  - **Indirectly Lost** leaks
  - **Possibly Lost** leaks (as they may really be 'Definitely Lost')
  - **Still Reachable** leaks (if really needed to fix them)
- When fixing 'Directly Lost' leaks, a single iteration is usually not needed. Once some of these leaks are fixed, newer 'Directly Lost' leaks may appear, and thus, new 'Directly Lost' fixes may be needed:
  - When removing a 'Directly Lost' leak, making the `start-pointer` to the block reachable again, all the 'Indirectly Lost' leaks that were chained to that original leak will become 'Indirectly Reachable'(includng those linked with 2 steps or more in a pointer chain).
  - When removing a 'Directly Lost' leak, freeing the corresponding block of memory, all the 'Indirectly Lost' leaks that were directly chained to that original leak will become 'Directly Lost' (not those linked with 2 or more steps in a pointer chain, which will remain 'Indirectly Lost').
- If the output report doesn't include a single 'Directly Lost' leak, it will also not show any 'Indirectly Lost' one.
- Once the programmer has decided the validity of the `interior-pointers` in cases 5 to 9, they will fall into one of the categories shown when tracking `start-pointers` (cases 1 to 4). For example, if both `interior-pointers` in Case 8 are valid ones, it should then be treated as Case 2 (AAA is 'Directly Reachable' and BBB is 'Indirectly Reachable').
- For different internal cases in `Memcheck`, the output report may show exactly the same information (as in Cases 7 and 8; or Cases 4 and 9).
- A side effect of having 'Possibly Lost' leaks is that, if the leaked blocks store pointers (start or interior ones) to other blocks, those will also get leaked as 'Possibly Lost'; and the output of `Valgrind` will not show any explicit link between them.